NEW SYLLABUS
CBCS PATTERN

B.B.A.
(Computer Application)
Semester-I

C PROGRAMMING

Dr. SUNITA D. PATIL
SALAUDDIN S. SAJJAN
KAMIL AJMAL KHAN

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

Syllabus ...

1. Introduction to C Language **[03]**

- 1.1 History
- 1.2 Basic Structure of C Programming
- 1.3 Language Fundamentals
  - 1.3.1 Character set, Tokens
  - 1.3.2 Keywords and Identifiers
  - 1.3.3 Variables and Data Types
- 1.4 Operators
  - 1.4.1 Types of Operators
  - 1.4.2 Precedence and Associativity
  - 1.4.3 Expression

2. Managing I/O Operations **[03]**

- 2.1 Console Based I/O and Related Built-in I/O Functions
  - 2.1.1 printf(), scanf()
  - 2.1.2 getch(), getchar()
- 2.2 Formatted Input and Formatted Output

3. Decision-Making and Looping **[06]**

- 3.1 Introduction
- 3.2 Decision-making Structure
  - 3.2.1 If Statement
  - 3.2.2 If-else Statement
  - 3.2.3 Nested if-else Statement
  - 3.2.4 Conditional Operator
  - 3.2.5 Switch Statement
- 3.3 Loop Control Structures
  - 3.3.1 while loop
  - 3.3.2 Do-while Loop
  - 3.3.3 For Loop
  - 3.3.4 Nested for Loop
- 3.4 Jump Statements
  - 3.4.1 break
  - 3.4.2 continue
  - 3.4.3 goto
  - 3.4.4 exit

4. Programs Through Conditional and Looping Statements **[06]**

- Addition / Multiplication of Integers
- Determining if a Number is +ve / -ve / even / odd
- Maximum of 2 Numbers, 3 Numbers
- Sum of First n Numbers, give n Numbers
- Integer Division, Digit Reversing, Table generation for n, ab
- Factorial, Sine, Series, Cosine series, nCr, Pascal Triangle, Prime number, Factors of number
- Other problems such as Perfect number, GCD of 2 numbers etc. (Write Algorithms and Draw Flowcharts)

[12]

5. Arrays and Strings

5.1 Introduction to one-dimensional Array
- 5.1.1 Definition
- 5.1.2 Declaration
- 5.1.3 Initialization

5.2 Accessing and displaying array elements
5.3 Finding smallest and largest number from array
5.4 Reversing array
5.5 Finding odd/even/prime number from array
5.4 Introduction to two-dimensional Array
- 5.4.1 Definition
- 5.4.2 Declaration
- 5.4.3 Initialization

5.5 Accessing and displaying array elements
5.6 Matrices: Addition, Multiplication, Transpose, Symmetry, Upper/Lower Triangular
5.7 Introductions to Strings
- 5.7.1 Definition
- 5.7.2 Declaration
- 5.7.3 Initialization

5.8 Standard library functions
5.9 Implementations without standard library functions. **[09]**

6. Functions

6.1 Introduction
- 6.1.1 Purpose of function
- 6.1.2 Function definition
- 6.1.3 Function declaration
- 6.1.4 Function call

6.2 Types of functions
6.3 Call by value and call by reference
6.4 Storage classes **[04]**

7. Introduction to Pointer

7.1 Definition
7.2 Declaration
7.3 Initialization
7.4 Indirection operator and address of operator
7.5 Pointer arithmetic
7.6 Dynamic memory allocation
7.7 Functions and pointers **[04]**

8. Structures

8.1 Introduction to structure
8.2 Definition
8.3 Declaration
8.4 Accessing Members
8.5 Structure Operations
8.6 Nested Structure

■■■

Contents ...

1...

Introduction to C Language

Learning Objectives...

- To study Introduction of C language.
- To study Fundamentals of C language.
- To understand various operators of C language and its applications.

1.1 INTRODUCTION TO C LANGUAGE [W-15, S-17]

A structured programming language offers a variety of programming possibilities and capabilities. It support different control loops, different statements etc.

C language is a general purpose and structured programming language developed by 'Dennis Ritchie' at AT&T's Bell Laboratories in the 1972s in USA.

C language combines the best elements of high-level language with control and flexibility of assembly language so it is a middle level language.

C language is also called as 'Procedure oriented Programming Language.'

C is a Middle level languages that don't provide all the built-in functions found in high level languages, but provides all building blocks that we need to produce the result we want.

C language allows the programmer to write low level programs as well as high level programs so it is called as Middle Level Language.

C has now become a widely used professional language for following reasons:

1. C language is easy to learn.
2. C language is structured language.
3. C language produces efficient programs.
4. C language can handle low-level activities.
5. C language can be compiled on a variety of computers.

1.1.1 Features of C

- C language consist of following features:

1. **Middle Level Language:** C is a middle level language as it combines elements of high-level language with the functional of assembly language. C allows direct manipulation of bits, bytes, words, and pointers.
2. **Block Structured Language:** C is referred as a structured language because it is similar in many ways to other structured languages like ALGOL, Pascal and the likes. C allows categorization of code and data. This is a distinguishing feature of any structured language.
3. **Code Portability:** The code written in C is machine independent which means, there is no change in 'C' instructions, when you change the Operating System or Hardware. There is hardly any change required to compile when you move the program from one environment to another.
4. **Recursion:** A function may call itself again and again this feature is called as recursion, is supported by C.
5. **Efficiency:** C provides fast program execution.
6. **High level language feature:** This feature allows the programmer to concentrate on the logic flow of the code.
7. **Low level features:** C has a close relationship with assembly languages. So it is easy to make assembly program in C.
8. **Powerful:** C is very powerful language since low level commands have been access like assembly language.
9. **Flexibility:** In C language, programmer has many ways to accomplish the same task.
10. **Small size:** C language provides no input output facilities. This helps to keep program small.
11. One important feature of C program, is its ability to extend itself.

1.1.2 Advantages

- Various advantages of C language are:

1. C is a building block for many other currently known languages.
2. It is Portable Programming Language. This means a program written for one computer may run successfully on other computer also.
3. It is fast for executing. This means that the executable program obtained after compiling and linking runs very fast.
4. It is Compact Programming Language. The statements in C Language are generally short but very powerful.
5. It is simple/easy. The C Language has both the simplicity of High Level Language and the speed of Low Level Language. So it is also know as Middle Level Language.

6. It has only 32 keyword so that are easy to remember.
7. Its compiler is easily available.
8. It has ability to extend itself. Users can add their own functions to the C Library.
9. It can use to design middle type of software.

1.1.3 Disadvantages

- Various disadvantages of C language are:
  1. There is no run-time checking.
  2. There is no strict type checking, (for example: We can pass an integer value for floating data type).
  3. As the program extends it is very difficult to fix the bugs.
  4. It may be compile time overhead due to the misplacing and excessive use of pointers.
  5. Now-a-days, it does not use to develop complex type of softwares.

1.1.4 Algorithm [W-14]

- A sequential solution of any program that written in human language, called Algorithm.
- In programming, algorithm is the set of well defined instruction in sequence to solve a program. Algorithm is first step of the solution process. After the analysis of problem, programmer writes the algorithm of that problem.
- Example: Algorithm to find out number is odd or even?

```
Step 1 : Start
Step 2 : input number
Step 3 : rem=number mod 2
Step 4 : if rem=0 then
         print "number even"
         else
         print "number odd"
         endif
Step 5 : Stop
```

1.2 HISTORY OF 'C' LANGUAGE

- The development of C was a cause of evolution of programming languages like ALGOL 60, CPL (Combined Programming Language), BCPL (Basic Combined Programming Language) and B.
- Fig. 1.1 shows brief history of C language.
- C was developed by Dennis Ritchie at Bell Laboratories in 1972. Most of its principle and ideas were taken from the earlier language B, BCPL and CPL.
- CPL was developed jointly between the Mathematical Laboratory at the University Cambridge and the University of London Computer Unit in 1960s.

- CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both machine independent programming and would allow the programmer to control the behavior of individual bits of information. But the CPL was too large for use in many applications.
- In 1967, BCPL (Basic Combined Programming Language) was created as a scaled down version of CPL while still retaining its basic features. This process was continued by Ken Thompson. He made B Language during working at Bell Labs.
- B Language was a scaled down version of BCPL. B Language was written for the systems programming.
- In 1972, a co-worker of Ken Thompson, Dennis Ritchie developed C Language by taking some of the generality found in BCPL to the B language.
- The original PDP-11 version of the Unix system was developed in assembly language.
- In 1973, C language had become powerful enough that most of the Unix kernel was rewritten in C. This was one of the first operating system kernels implemented in a language other than assembly.
- In 1978, Dennis Ritchie and Brian Kernighan published the first edition "The C Programming Language" and commonly known as K&R C.
- The 1989 standard for C is commonly referred as C89.
- During 1990s, the development of C++ language consumed most programmers attention but work on C continued quietly along with a new standard for C being developed and the result was the 1999 standard for C, usually referred as C99. It retained all features of C89.
- In 2007, work began on another revision of the C standard, called C11 (formerly C1X) until its official publication on 2011. C11 mainly standardizes features already supported by common contemporary compilers, and includes a detailed memory model to better support multiple threads of execution.
- In June 2018, C18 is the current standard for the C Programming Language.

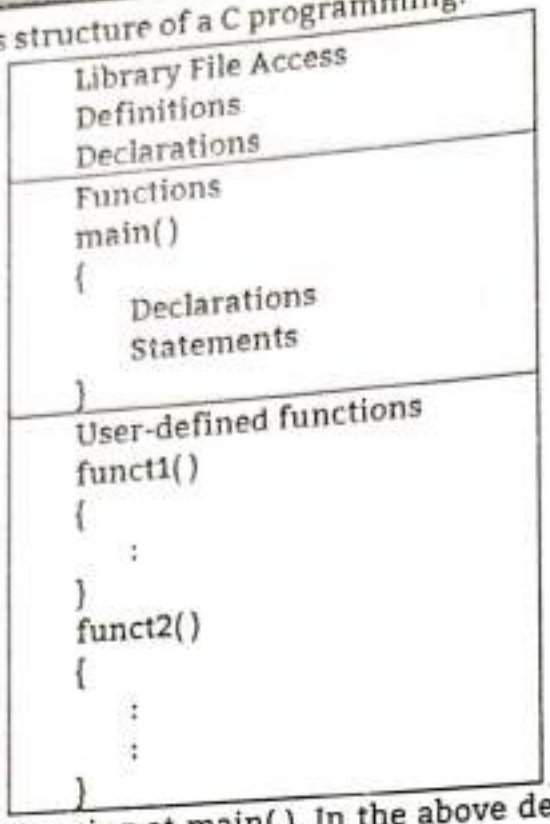
Year of establishment	Language name	Description
1960	ALGOL-60 ⇓	ALGOL is an acronym for Algorithmic Language developed by Cambridge University. It was the first structured procedural programming language, developed in the late 1950s and once widely used in Europe. But it was too abstract and too general structured language.
1963	CPL (Combined Programming Language) ⇓	CPL is an acronym for Combined Programming Language. It was developed at Cambridge University.

1967	BCPL (Basic Combined Programming Language) ⇓	BCPL is an acronym for Basic Combined Programming Language. It was developed by Martin Richards at Cambridge University in 1967. BCPL was not so powerful. So, it was failed.
1970	B ⇓	B language was developed by Ken Thompson at AT & T Bell Laboratories in 1970. It was machine dependent. So, it leads to specific problems.
1972	C ⇓	'C' Programming Language was developed by Dennis Ritchie at AT&T Bell Laboratories in 1972. This is general purpose, compiled, structured programming language. Dennis Ritchie studied the BCPL, then improved and named it as 'C' which is the second letter of BCPL.
1978	K&R C ⇓	Dennis Ritchie and Brian Kernighan published the first edition "The C Programming Language" and commonly known as K&R C.
1989	C89/C90 standard ⇓	First standardized specification for C language was developed by American National Standards Institute in 1989. C89 and C90 standards refer to the same programming language.
1999	C99 standard	Next revision was published in 1999 that introduced new futures like advanced data types and other changes. It retained nearly all of the features of C89, thus C is still C.
2011	⇓ C11	The C11 standard adds numerous new features to C and the library, including type generic macros, anonymous structures, improved Unicode support, atomic operations, multi-threading, and bounds-checked functions.
2018	⇓ C18	It introduces no new language features, only technical corrections and clarifications to defects in C11.

Fig. 1.1: Brief History of C language

1.3 BASIC STRUCTURE OF C PROGRAMMING [S-16]

- Following blocks shows structure of a C programming.



- The C program begin executing at main( ). In the above declaration, library files are used to give instructions to compiler for linking purpose.
- All constants and global variables declarations is done here. In second box consists of main( ). Local variable declarations and statements defined in main( ).
- The last and third part consists of all user defined functions. These functions are called in main( ) function.

A Simple C Program ("Hello World"):

- Every C program must contain main( ) function. Execution of each and every C program starts with main function only.
- Every C program can contain more than one function but each program has to be a main( ) function in order to execute the program.
- C program is also called as **Procedure Oriented Programming Language** where importance is given to the procedure i.e. the task to be done which is expressed in the form of functions. So functions works as a building block for the programs.
- The program prints out "Hello World" to the standard output, which is usually a terminal or screen display.

```
1. #include<stdio.h>
2. int main(void)
3. {
4. printf("Hello World\n");
5. return 0;
6. }
```

Explanation of 'C' program are given below:

1. **#include<stdio.h>**

   This first line of the program is a preprocessing directive, #include. The #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found. Include directives are typically used to include the C header files for C functions that are held outsite of the current source file. The header file **stdio.h** contains declarations for standard input and output functions such as **printf**.

2. **int main(void)**

   This line indicates that a function named 'main' is being defined. The main function serves a special purpose in C programs. The run-time environment calls the main function to begin program execution. The type specifier int indicates that the return value, (the value of evaluating the main function) is returned to its invoker (in this case the run-time environment), is an integer. The keyword void as a parameter list indicates that the main function takes no arguments.

3. **{**

   This opening curly brace indicates the beginning of the definition of the main function.

4. **printf("Hello World\n");**

   This line calls (executes the code for) a function named **printf**, which is declared in the included header **stdio.h** and supplied from a system library. In this call, the printf function is passed (provided with) a single argument, the address of the first character in the string literal "Hello World\n". The semicolon (;) terminates the statement.

5. **return 0;**

   This line terminates the execution of the main function and causes it to return the integer value 0, which is interpreted by the run-time system as an exit code, (indicating successful execution).

6. **}**

   This closing curly brace indicates the end of the code for the main function.

Compile and Execute C Program:

- **Compilation is the process of converting a C program which is user readable code into machine readable code which is 0s and 1s.**
- **This compilation process is done by a compiler which is an inbuilt program in C.**
- **As a result of compilation, we get another file called executable file. This is also called as binary file.**
- **This binary file is executed to get the output of the program based on the logic written into it.**

Steps to Compile and Execute a C Program:

Step 1 : Type the above C basic program in a text editor and save as "sample.c".

Step 2 : To compile this program, open the command prompt and goto the directory where you have saved this program and type "cc sample.c" or "gcc sample.c".

Step 3 : If there is no error, executable file will be generated in the name of "a.out".

Step 4 : To run this executable file, type "./a.out" in command prompt.

Step 5 : You will see the output as shown in the above basic program.

1.4 LANGUAGE FUNDAMENTALS

1.4.1 Character Set

- Character set are the set of alphabets, letters and some special characters that are valid in C language.
- A character refers to the digit, alphabet or special symbol used to data representation.
- The C character set consists of all uppercase characters A to Z, the lowercase characters a to z, the digits 0 to 9, certain special characters and white spaces.
- The special character are listed below:

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

- White-space characters:

backspace	\b	Vertical tab	\v
newline	\n	form feed	\f
horizontal tab	\t	carriage return	\r

- These characteristics combinations are known as escape sequence.

1.4.2 Tokens [S-17]

- In a C program, the smallest individual meaningful units is called token.
- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Fig. 1.2 shows various tokens in C language.

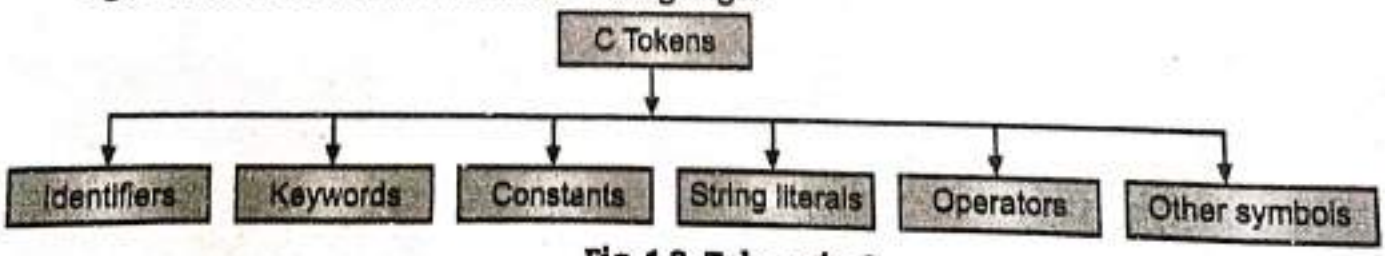


Fig. 1.2: Tokens in C

1. Identifiers are user defined words like sum, roll_no, sub_marks and used for variable and function names.
2. Keywords are reserved words like int, for, while etc.
3. Constants are fixed values which does not change like 10, S etc.
4. Operators are symbols which represents an operation like +, −, * etc.
5. Other symbols are symbols which have particular meaning like; (semicolon) indicates end of an instruction.
6. String literals is a sequence of zero or more characters enclosed in double quotes like "Nirali Prakashan".

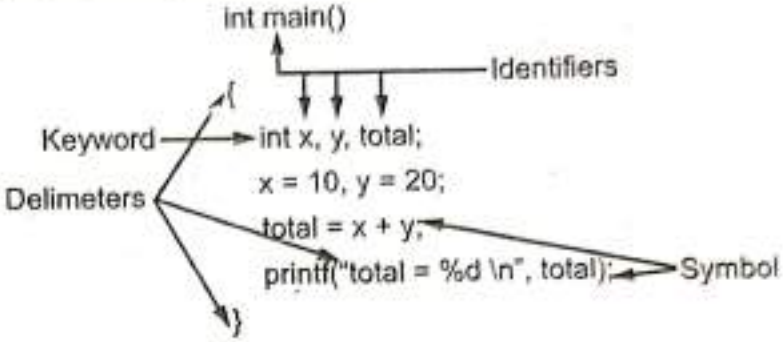


Fig. 1.3: Various tokens in program

- Fig. 1.3 shows various tokens in C program.

1.4.3 Keywords

[W-15, S-18]

- Keywords are the reserved words used in C programming. Each keyword has fixed meaning and that cannot be changed by user.
- Keywords are standard identifiers that have standard predefined meaning in C.
- It is strongly recommended that Keywords should be in lower case letters.
- Keywords can be used only for their intended purpose. Keywords can't be used as programmer defined identifier.The keywords can't be used as names for variables.
- There are totally **32 keywords** used in a C programming.

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

1.4.4 Identifiers

[W-14, S-16]

- An identifier is a user defined name used to represent program elements such as function names, variables, structures etc.
- Identifier is the name of a variable that is made up from combination of alphabets, digits and underscore.
- Identifiers are created to give unique name to C entities to identify it during the execution of program.

The rules which should be followed while naming an identifier are:

1. Identifier name must be a sequence of letter and digits, and must begin with a letter.
2. The underscore character ('_') is considered as letter.
3. Names shouldn't be a Keyword (such as int , float, if ,break, for etc)
4. Both upper-case letter and lower-case letter characters are allowed. However, they're not interchangeable.
5. No identifier may be Keyword.
6. No special characters, such as semicolon,period,blank space, slash or comma are permitted

Examples of valid identifiers:

```
sum, sum1, price_of_item, Rate_of_interest, add_odd.
```

Examples of invalid identifiers:

```
2rate, 6a, a + b, x%y
```

- The length of the identifier can be arbitrarily long but most of the compilers in C recognize only first eight characters.

1.4.5 Constants

- Constants is a fixed value which do not change during program execution.
- Constants are also refer as literals.
- For a particular program we can have certain constants (For example: for calculating area of circle, pi = 3.148 is a constant) so instead of storing them in a particular variable and then using them, we can directly use them.
- Constants can be of any of the basic data type.
- Constants are the terms that can't be changed during the execution of a program. For example: 1, 2.5, "Programming is easy." etc.
- In C, constants can be classified as follows:

1. **Integer Constants:**

- Integer constants are the numeric constants (constant associated with number) without any fractional part or exponential part. There are three types of integer constants in C language decimal constant (base 10), octal constant (base 8) and hexadecimal constant (base 16).
  - Decimal digits: 0 1 2 3 4 5 6 7 8 9.
  - Octal digits: 0 1 2 3 4 5 6 7.
  - Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F.

  For example:
  - Decimal constants: 0, -9, 22 etc.
  - Octal constants: 021, 077, 033 etc.
  - Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

2. **Floating Point Constants:**

- Floating point constants requires the decimal point followed by the number's

- We can represent exponential data as constant using floating point constant.
- Floating point constant have following rules:
  (i) They have a decimal point and digits from 0 to 9.
  (ii) Commas and blank spaces are not permitted.
  (iii) They can be negative.
  (iv) It is possible to omit digits before or after the decimal point.

  Examples:
  (i) 2×10^7 can be present in many ways:
  20000000 2e7 2e 7 2E7 2.0e + 7
  0.2e8 .2E8
  (ii) 3.2×10^{-6} can be represent as:
  3.2 E – 6 3.2e – 6 etc.

3. Character Constants:

- A character constants can contain or store only a single character enclosed in apostrophes (single quotation marks).
  Example: 'x' 'a' 'M' '6' '$'
- The value of the character constant is the corresponding numeric value of the character. So, we can compare the two characters using numeric as ASCII value.

4. String Constants:

- A string constant is a set of characters enclosed in double quotes.
- C allows us to define string constants, it does not formally have a string data type.

Example:
```
"Let us learn C"
"123 - 45"
"Welcome to \n BCA Semester II"
```

Backslash Characters/Escape Sequences: [W-14, S-17]

- C supports some special escape sequence characters that are used to do special tasks. These are also called as 'Backslash characters'.
- Table 1.1 shows blackslash characters/escape sequences.

Table 1.1

Character constant	Meaning
\n	New line (Line break)
\b	Backspace
\t	Horizontal Tab
\f	Form feed
\a	Alert (alerts a bell)
\r	Carriage Return
\v	Vertical Tab
\?	Question Mark
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null

1.4.6 Variables

- A variable is a name assigned to the memory location where data is stored. In other words, variable is the data name that refers to the stored value.
- Variables are memory location in computer's memory to store data. To indicate the memory location, each variable should be given a unique name called identifier.
- Fig. 1.4 shows a variable num, with value 10 and memory address (location 2000).

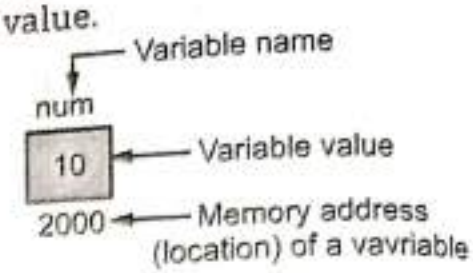


Fig. 1.4: Variables

Rules for naming variable:

1. Variable name must begin with letter or underscore.
2. Variables are case sensitive.
3. They can be constructed with digits, letters.
4. No special symbols are allowed other than underscore.
5. The variable name can contain maximum 31 characters.

1.4.6.1 Declaring and Initializing Variables

- Before use variables should be declared in the C program. Memory space is not allocated for a variable while declaration. It happens only on variable definition.
- Variable initialization means assigning a value to the variable.

Type	Syntax	Example
1. Variable declaration	`data_type variable_name;`	`int x, y, z; char flat, ch;`
2. Variable initialization	`data_type variable_name = value;`	`int x = 50, y = 30; char flag = 'x', ch='1';`

Difference between Variable Declaration and Definition:

Variable declaration	Variable definition
1. Declaration tells the compiler about data type and size of the variable.	1. Definition allocates memory for the variable.
2. Variable can be declared many times in a program.	2. It can happen only one time for a variable in a program.
3. The assignment of properties and identification to a variable.	3. Assignments of storage space to a variable.

1.4.6.2 Types of Variables

- There are two types of variables i.e., local and global variables.

1. Local Variables:

- Variables that are declared inside a function are called local variables.

- Local variables can be used only by the statement which is inside the block and in which variables are declared.
- Local variables cannot be used outside the block. Lifetime of the local variable is till the end of the block i.e. a local variable is created when block enters and destroyed when block exit.
- We can declare same variable name within two different blocks. However, we cannot have same variable name within one block.

Program 1.1: Program for local variables.

```
#include <stdio.h>
int main()
{
    int m=40,n=20;      // m, n are local variables
    if (m == n)
    {
       printf("m and n are equal");
    }
    else
    {
       printf("m and n are not equal");
    }
    return 0;
}
```

Output:

```
m and n are not equal
```

Formal Parameters: [W-14]

- If function use arguments, it must declare variables so that the variables will accept the values of the arguments. These variables are referred as formal parameters of the function.
- Formal parameters are local to function.

Example:

```
double addition(int x, int y)
{
    double Z;
    Z = x + y;
    return Z;
}
```

- The function addition has two parameters x and y. This function returns addition x and y.

2. Global Variables:

- Global variables are used anywhere in the program and they will hold their values throughout the program execution.
- Global variables are declared outside of the function.

Program 1.2: Program for global variables.

```
#include <stdio.h>
int n=20;                    // n is global variable
int main()
{
    int m=40;                // m is local variable
    if (m == n)
    {
        printf("m and n are equal");
    }
    else
    {
        printf("m and n are not equal");
    }
    return 0;
}
```

Output:

```
m and n are not equal
```

[S-18]

- Normally, global variables are declared at the top of the program. Global variables are helpful when many functions in your program use the same data.
- Global variables are stored in a fixed region of memory and they take up memory the entire time your program is executing, not just when they are needed.

1.4.7 Data Types

[S-15]

- Data types are the keywords, which are used for assigning a type to a variable.
- **Definition:** "The data storage format that a variable can store a data to perform a specific operation".

OR

- Data type can be defined as "the type of data of variable or constant store."
- Data types are used to define a variable before to use in a program. Size of variable, constant and array are determined by data types.
- When we use a variable in a program then we have to mention the type of data. This can be handled using data type in C.

- Fig. 1.5 shows datatypes in C language.

1. **Primitive data types** are the first form – the basic data types (int, char, float, double).
2. **Derived data types** are a derivative of primitive data types known as arrays, pointer and function.
3. **User defined** data types are those data types which are defined by the user/programmer himself.

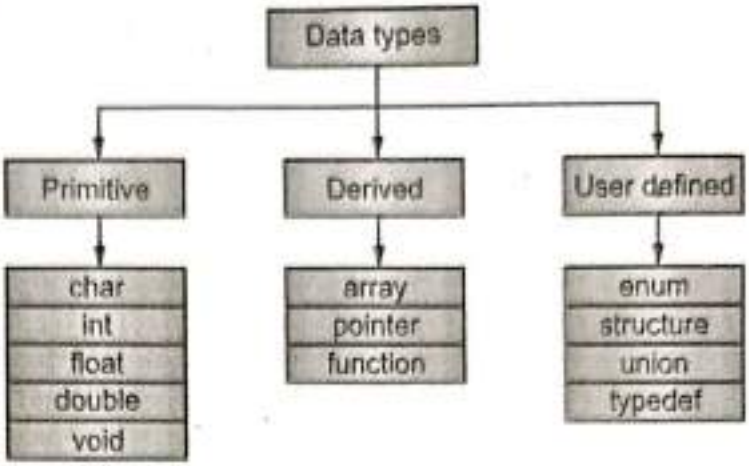


Fig. 1.5: Datatypes in C Language

1. Simple / Built-in / Fundamental Data Types:

- Built-in Fundamental data types are atomic types. Built-in types are also referred as **'Primitive'** data types.
- Table 1.2 shows the most commonly used data types in C.

Table 1.2

Keyword	Format (Access) Specifier	Size	Data Range
char	%c	1 Byte	-128 to +127
unsigned char	%c	8 Bytes	0 to 255
int	%d	2 Bytes	-32768 to +32767
long int	%ld	4 Bytes	-231 to +231
unsigned int	%u	2 Bytes	0 to 65535
float	%f	4 Bytes	-3.4e38 to +3.4e38
double	%lf	8 Bytes	-1.7e38 to +1.7e38
long double	%Lf	12-16 Bytes	-3.4e38 to +3.4e38

(i) **Integer:** Integer data type is simply written as **int**. To store the integer i.e. whole number, typical memory requirement is 1 word or 2 bytes, (1 byte consists of 8 bits). They can be further of the type: long int, short int, signed int, unsigned int, these are called as **qualifiers.**

(ii) **Character:** This is used when we deal with character type of data containing a – z or A – Z characters available in C character set.

(iii) **Float:** It is also used to store numeric data only the difference is that we can use numeric data having decimal values. Some compilers permits use of long, float, short float etc.

(iv) **Double:** If we use exponential in the programme then it forms a very long number. So we use type double.

(v) **Void:** Using void data type, we can specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function. Void is empty data type.

Example:

(i) `void func1 (int x)`

Here, func1 does not return any value.

(ii) `int func2 (void)`

We can use void as a data type as parameter list.

2. **Derived Data Types:**

- The data types which are derived from built-in or primitive data types are called as derived data types.
- These data types can be derived using declaration operators are punctuators.

(i) **Array:** An array is a derived data type as it can be derived from built-in data types.

It is basically a named, consecutive and finite set of memory locations, which can hold multiple data items of its declared data type.

Example:

- `int ARR[5], /*Here ARR is an array of type integer having 5(0 to 4) consecutive memory blocks*/`
- `float ARR[5], /*Here ARR is an array of type floating-point having 5(0 to 4) consecutive memory blocks*/`

(ii) **Functions:** A function in C is a named and finite set of C program statements. They are designed to do some specific task or to take some specific actions. Functions are associated with return types, actual and formal argument list. Typically a function is associated with three tasks.

- Function prototyping,
- Function definition writing, and
- Function calling.

(iii) **Pointers:** A pointer is a special variable which holds the address of a normal variable of its own data type. The pointer variable can be declared with the help of a punctuator *(astrick).

Example:

```
int *ptr,     /*declared a pointer variable named as 'ptr' and is
              of type 'int' */
int num;
ptr = &num;   /*assigned the address of variable 'num' of type
              'int' to the pointer variable */
```

3. User Defined Types:

- User defined data types are the customized data type. It can be composed of variety of the built-in and derived data types (excluding function).
- A user-defined data type is specially designed to meet once programming needs.
- In C language, the user defined data types are created by using following data types.

(i) **Enum Data Type:** This is an user defined data type having finite set of enumeration constants. The keyword 'enum' is used to create enumerated data type.

Syntax:
```
enum [data_type] {const1, const2, ...., const n};
```
Example:
```
enum mca(software, web, seo);
```

(ii) **typedef:** It is used to create new data type. But it is commonly used to change existing data type with another name.

Syntax:
```
typedef [data_type] synonym;
```
OR
```
typedef [data_type] new_data_type;
```
Example:
```
typedef int integer;
integer rno;
```

(iii) **Structure:** A structure is composed of one or more built-in or/and derived data types (excluding function). Every element of the structure has separate storage space. So the size of a variable of structure type will have the memory space equal to the sum of memory spaces of its all individual elements. Keyword struct is used define a structure.

(iv) **Union:** A union is also composed of one or more built-in and/or derived data types (excluding function). Keyword union is used to define a unions.

1.5 OPERATORS [W-14, 16, S-16]

- Operators are the symbol which operates a value or a variable.
- Operators represent an operation.
- The symbols which are used to perform logical and mathematical operations are called operators.
- C is very rich in built-in operations. Operators instruct the compiler to perform some actions on operands.
- An operator can be unary (one operand), binary (two operands) or ternary (three operand).

Example:

unary	−5
binary	3 + 2
ternary	conditional operators
	(a > b) ? a: b

1.5.1 Types of Operators

1. Arithmetic Operators:

- It is also called as 'binary operators'. It is used to perform arithmetical operations. These operators operate on two operands.
- Following are arithmetic operators supported by C language and assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands.	A + B will give 30
-	Subtracts second operand from the first.	A - B will give -10
*	Multiply both operands.	A * B will give 200
/	Divide numerator by enumerators.	B / A will give 2
%	Modulus Operator and remainder of after an integer division.	B % A will give 0
++	Increment operator, increases integer value by one.	A++ will give 11
--	Decrement operator, decreases integer value by one.	A-- will give 9

Program 1.3: Program for arithmetic operators.

```
#include <stdio.h>
void main()
{
int a = 21;
int b = 10;
int c ;
c = a + b;
printf("Value of c is %d\n", c );
   c = a - b;
   printf("Value of c is %d\n", c );
   c = a * b;
   printf("Value of c is %d\n", c );
   c = a / b;
   printf("Value of c is %d\n", c );
   c = a % b;
   printf("Value of c is %d\n", c );
   c = a++;
   printf("Value of c is %d\n", c );
   c = a--;
   printf("Value of c is %d\n", c );
}
```

Output:

```
Value of c is 31
Value of c is 11
Value of c is 210
Value of c is 2
Value of c is 1
Value of c is 21
Value of c is 22
```

2. Relational and Logical Operators: [W-14, 15]

- Relational operators used in expressions and result of expression are either true (1) or false (0).
- Logical operators are used to perform logical operations on the given two variables.
- There are following relational operators supported by C language. Assume variable A holds 10 and variable B holds 20 then:

Relational Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is false.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is false.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is false.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

- Following table shows various logical operations.

Logical Operator	Description	Example
&& (Logical AND operator)	If both the operands are non zero then condition becomes true.	(A && B) is true.

contd. ...

|| (Logical OR Operator)	If any of the two operands is non zero then condition becomes true.	(A || B) is true.
! (Logical NOT Operator)	Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

Program 1.4: Program for relational operators.

```
#include <stdio.h>
void main()
{
int a = 21;
int b = 10;
   int c ;
if( a == b )
{
   printf("a is equal to b\n" );
}
else
{
   printf("a is not equal to b\n" );
}
if ( a < b )
{
   printf("a is less than b\n" );
}
else
{
   printf("a is not less than b\n" );
}
if ( a > b )
{
   printf("a is greater than b\n" );
}
else
{
   printf("a is not greater than b\n" );
}
```

```
/* Lets change value of a and b */
a = 5;
b = 20;
if ( a <= b )
{
   printf("a is either less than or equal to  b\n" );
}
   if ( b >= a )
{
   printf("b is either greater than  or equal to a\n" );
}
}
```

Output:

```
a is not equal to b
a is not less than b
a is greater than b
a is either less than or equal to  b
b is either greater than  or equal to a
```

Program 1.5: Program for logical operators.

```
#include <stdio.h>
void main()
{
   int a = 5;
   int b = 20;
   int c ;
   if ( a && b )
   {
      printf("Condition is true\n" );
   }
   if ( a || b )
   {
      printf("Condition is true\n" );
   }
   /* lets change the value of  a and b */
   a = 0;
   b = 10;
   if ( a && b )
   {
      printf("Condition is true\n" );
   }
```

```
    else
    {
        printf("Condition is not true\n" );
    }
    if ( !(a && b) )
    {
        printf("Condition is true\n" );
    }
}
```

Output:

```
Condition is true
Condition is true
Condition is not true
Condition is true
```

3. Assignment Operators:

- These operators are used to assign a value to variable.
- There are following assignment operators supported by C language:

Operator	Description	Example
= (Simple assignment operator)	Assigns values from right side operands to left side operand.	C = A + B will assigne value of A + B into C
+= (Add AND assignment operator)	It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-= (Subtract AND assignment operator)	It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*= (Multiply AND assignment operator)	It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/= (Divide AND assignment operator)	It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A

contd...

%= (Modulus AND assignment operator)	It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Program 1.6: Program for assignment operators.

```
#include <stdio.h>
void main()
{
   int a = 21;
   int c ;
   c =  a;
   printf("Value of c = %d\n", c );
   c +=  a;
   printf("Value of c = %d\n", c );
   c -=  a;
   printf("Value of c = %d\n", c );
   c *=  a;
   printf("Value of c = %d\n", c );
   c /=  a;
   printf("Value of c = %d\n", c );
   c  = 200;
   c %=  a;
   printf("Value of c = %d\n", c );
   c <<=  2;
   printf("Value of c = %d\n", c );
   c >>=  2;
   printf("Value of c = %d\n", c );
   c &=  2;
   printf("Value of c = %d\n", c );
   c ^=  2;
```

```
    printf("Value of c = %d\n", c );
    c |= 2;
    printf("Value of c = %d\n", c );
}
```

Output:

```
Value of c = 21
Value of c = 42
Value of c = 21
Value of c = 441
Value of c = 21
Value of c = 11
Value of c = 44
Value of c = 11
Value of c = 2
Value of c = 0
Value of c = 2
```

[W-14]

4. Bitwise Operators:

- Bitwise operator works on bits and performs bit by bit operation.
- Bitwise operators are used for manipulation of data at a bit level.
- They can be directly applied to char, short int and long.
- Assume if A = 60; and B = 13; Now in binary format they will be as follows:

  A = 0011 1100
  B = 0000 1101

  A&B = 0000 1100
  A|B = 0011 1101
  A^B = 0011 0001
  ~A = 1100 0011

- There are following Bitwise operators supported by C language.

Operator	Description	Example
& (Binary AND Operator)	Copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (Binary OR Operator)	Copies a bit if it exists in eather operand.	(A | B) will give 61 which is 0011 1101

contd. ...

^ (Binary XOR Operator)	Copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (Binary Ones Complement Operator)	Is unary and has the efect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<< (Binary Left Shift Operator)	The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (Binary Right Shift Operator)	The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Program 1.7: Program for bitwise operators.

```
#include <stdio.h>
void main()
{
   unsigned int a = 60;   /* 60 = 0011 1100 */
   unsigned int b = 13;   /* 13 = 0000 1101 */
   int c = 0;
   c = a & b;        /* 12 = 0000 1100 */
   printf("Value of c is %d\n", c );
   c = a | b;        /* 61 = 0011 1101 */
   printf("Value of c is %d\n", c );
   c = a ^ b;        /* 49 = 0011 0001 */
   printf("Value of c is %d\n", c );
   c = ~a;           /*-61 = 1100 0011 */
   printf("Value of c is %d\n", c );
   c = a << 2;     /* 240 = 1111 0000 */
   printf("Value of c is %d\n", c );
   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Value of c is %d\n", c );
}
```

Output:

```
Value of c is 12
Value of c is 61
Value of c is 49
Value of c is -61
Value of c is 240
Value of c is 15
```

5. Increment and Decrement Operators

- Increment (++) and decrement (– –) operators are also unary operators.
- The operator ++ adds one to its operand, whereas the operator – – subtracts one from its operand.
- For justification, a = a + 1; can be written as a++; and a=a–1; can be written as a– –; Both of these operators may either follow or precede the operand. i.e. a = a + 1; can be represented as a++; or ++a;
- If '++' or '– –' are used as suffix to the variable name, then post increment/decrement operation take place. Consider an example for understanding '++' operator as a suffix to the variable.

```
a = 20;
b = 10;
c = a * b ++;
```

- In the above equation, the current value of 'b' is used for the product. The result is 200 which is assigned to 'c'. After multiplication, the value of 'b' is incremented by one.
- If '++' or '– –' are used as prefix to the variable name, then pre increment/decrement operation take place. Consider an example of understanding '++' operators as a prefix to the variable.

```
a = 20;
b = 10;
c = a *  ++ b;
```

- In the above equation, the value of 'b' is incremented and then multiplication is carried. The result is 220, which is assigned to 'c'.

Program 1.8: Program of increment and decrement operators.

```
#include<stdio.h>
int main()
{
    int a = 21;
    int c ;
    c = a++;
    printf("Value of c is %d\n", c );
    c = a--;
    printf("Value of c is %d\n", c );
    return 0;
}
```

Output:

```
Value of c is 21
Value of c is 22
```

6. Other Operators: [W-14]

- There are few other operators supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
?:	Conditional Expression	If Condition is true? Then value X: Otherwise value Y
,	Comma Operator	Used to separate set of expressions.

1. sizeof Operator:

- It is an unary operator which provides the size, in bytes, of the given operand.
- The syntax of sizeof operator is,

  Syntax: `sizeof (operand)`

  Here, the operand is a built-in or user-defined data type or variable.
- The sizeof operator always precedes its operand.

  For example: `sizeof(float)`

  returns the value 4.
- This information is quite useful when we execute our program on a different computer or a new version of C is used. The sizeof operator helps in case of dynamic memory allocation for calculating the number of bytes used by some user-defined data type.

Program 1.9: Program for sizeof operator.

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%d bytes\n",sizeof(a));
    printf("Size of float=%d bytes\n",sizeof(b));
    printf("Size of double=%d bytes\n",sizeof(c));
    printf("Size of char=%d byte\n",sizeof(d));
    return 0;
}
```

Output:

```
Size of int=2 bytes
Size of float=4 bytes
Size of double=8 bytes
Size of char=1 byte
```

2. comma Operator:

- The comma (,) operator is used to separate two or more expressions.
- The comma operator has the lowest priority among all the operators.
- It is not essential to enclose the expressions with comma operators within th parenthesis. For example the statements given below are valid.

For example:

```
a = 2, b = 4, c = a + b;
(a = 2, b = 3, c = a + b;)
```

Program 1.10: Program for comma (,) operator.

```
#include<stdio.h>
int main()
{
    int a, b;
    clrscr();
    a=1,2,3,4,5;
    b=(1,2,3,4,5);
    printf("Resultant values of a and b are:\n");
    printf("%d %d", a, b);
    return 0;
}
```

Output:

```
Resultant values of a and b are:
1 5
```

3. The & and * pointer operators:

- A pointer is the memory address of an object. The pointer operator & is a unar operator that returns the memory address of its operand.

  Example: `a = &x;`
- The memory address of variable x is places into a. This memory address is th computer's internal location of the variable. If x is stored at memory location 100 then value of 'a' becomes 1001.
- The pointer operator *, is the complement of &. It is an unary operator that returns th value of the object lr cated at the address that follows it.

1.5.2 Precedence and Associativity of Operators

1. Precedence of operators:

- If more than one operators are involved in an expression then, C language ha predefined rule of priority of operators. This rule of priority of operators is calle operator precedence.
- In C, precedence of arithmetic operators (*,%,/,+,-) is higher than relational operator (==,!=,>,<,>=,<=) and precedence of relational operator is higher than logica operators (&&, || and !).

2. Associativity of operators:

- Associativity indicates in which order two operators of same precedence (priority) executes. Let us suppose an expression:

```
a==b!=c
```

- Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e., the expression in left is executed first and execution take place towards right. Thus, a==b!=c equivalent to:

```
(a==b)!=c
```

- The Table 1.3 below shows all the operators in C with precedence and associativity.

Table 1.3

Operator	Meaning of Operator	Associativity
() [] -> .	Functional call Array element reference Indirect member selection Direct member selection	Left to right
! ~ + - ++ -- & * sizeof (type)	Logical negation Bitwise (1's) complement Unary plus Unary minus Increment Decrement Dereference Operator (Address) Pointer reference Returns the size of an object Type cast (conversion)	Right to left
* / %	Multiply Divide Remainder	Left to right
+ -	Binary plus (Addition) Binary minus (Subtraction)	Left to right
<< >>	Left shift Right shift	Left to right
< <= > >=	Less than Less than or equal Greater than Greater than or equal	Left to right
== !=	Equal to Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right

contd. ...

&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional Operator	Left to right
	Simple assignment	Right to left
=	Assign product	
*=	Assign quotient	
/=	Assign remainder	
%=	Assign sum	
-=	Assign difference	
&=	Assign bitwise AND	
^=	Assign bitwise XOR	
|=	Assign bitwise OR	
<<=	Assign left shift	
>>=	Assign right shift	
,	Separator of expressions	Left to right

1.5.3 Expressions

- An expression in C is a combination of constants, functions and variables written according to the syntax of language.
- We can define expression as "A series of variables, operators and constants calls that evaluates to a single value".
- Expressions perform the work of a program.
- Expressions are evaluated using precedence of operators.
- When the variables of different types are mixed in an expression, they are all converted to the same type.
- Examples of expressions in C are given below:

```
z = x + y
a = b
c = a + b;
a <= b;
c == d;
```

Summary

- C language is developed by 'Dennis Ritchie' at AT&T's Bell Laboratories in 1972 in USA.
- Every C program execution begins with main() function.
- The main function can call other standard functions as well as user defined functions.
- 'C' language is case sensitive i.e. it differentiates between uppercase and lowercase letters.

- Algorithm is sequential solution of any program that written in human language.
- Every C statement must end with a semicolon.
- C language fundamentals are Character Set, Tokens, Keywords, Identifiers, Constants, Variables, Data Types.
- Token is small individual unit in programs.
- C language has 32 keywords.
- In C, various types of operators like Arithmetic Operator, Logical Operator, Assignment Operator, Relational Operator, Bitwise Operator.

Check Your Understanding

1. Which of the following is not a valid variable name declaration?
   (a) int _a3; (b) int a_3;
   (c) int 3_a; (d) int _3a
2. All keywords in C are in _____
   (a) LowerCase letters (b) UpperCase letters
   (c) CamelCase letters (d) None of the mentioned
3. The format identifier '%i' is also used for _____data type.
   (a) char (b) int
   (c) float (d) double
4. Which of the following is a User-defined data type?
   (a) typedef int Boolean;
   (b) typedef enum {Mon, Tue, Wed, Thu, Fri} Workdays;
   (c) struct {char name[10], int age};
   (d) all of the mentioned
5. What is the precedence of arithmetic operators (from highest to lowest)?
   (a) %, *, /, +, – (b) %, +, /, *, –
   (c) +, -, %, *, / (d) %, +, -, *, /
6. What is the result of a logical or relational expression in C?
   (a) True or False
   (b) 0 or 1
   (c) 0 if an expression is false and any positive number if an expression is true
   (d) None of the mentioned
7. Which type of conversion is NOT accepted?
   (a) From char to int (b) From float to char pointer
   (c) From negative int to char (d) From double to char

Answers

1. (c)	2. (a)	3. (b)	4. (d)	5. (a)	6. (b)	7. (b)

Trace The Output

(i)
```
main()
{
    int x = 20 * 30, y;
    y = x/2
    printf("%d %d", x, y);
}
```

(ii)
```
main()
{
    int a, b, c;
    a = b = c = - 1
    cx = ++ d && ++b || c;
    printf ("a = %d, b = %d, c = %d, a, b, c);
}
```

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

(iv)
```
main()
{
    const int a;
    a = 130;
    printf("%d", a);
}
```

Practice Questions

Q.1 Write the answer of following questions in brief:

1. What is C language? Enlist its features.
2. Describe the header file and its usage in C programming?
3. What are the types of operators in C language.
4. Enlist basic data types is C language.
5. Enlist any eight keywords in C language.

Q.2 Write the answer of following questions:

1. State various advantages and disadvantages of C language.
2. Explain basic structure of C programming diagrammatically.
3. What are the types of operators in C language? Explain two of them in detail.
4. Write short note on: Precedence and associativity of operators.
5. Differentiate between global and local variables with example.

6. Explain history of C language.
7. Explain two operators in detail.
8. How to declare and define variable? Explain with example.

Q.3 Define terms.
(i) Identifier, (ii) Constant, (iii) Expressions, (iv) Token, (v) Keywords, (vi) Variable, (vii) Header file, (viii) Operator, (ix) Global variable, (x) Local variable.

Previous Exam Questions

Winter 2014

1. State different types of relational operators in 'C'. [2 M]

Ans. Refer to Section 1.5.

2. What is formal parameter? Give example. [2 M]

Ans. Refer to Section 1.4.6.2.

3. What is Identifier? [2 M]

Ans. Refer to Section 1.4.4.

4. Define Algorithm. [2 M]

Ans. Refer to Section 1.1.4.

5. What is escape sequence? [2 M]

Ans. Refer to Section 1.4.5.

6. [5 M]

```
main()
{
   int x, y, z;
   x = y = z = 1;
   z = + + x || + + y & & + + z;
   printf(" x = % d y = % d z = % d \ n", x, y, z);
}
```

Ans. **Output:** x = 2, y = 1, z = 1.

Summer 2015

1. List primary data type in C language. [2 M]

Ans. Refer to Section 1.4.7.

2. "Size of is an operator in 'C' state true to false. [2 M]

Ans. Refer to Section 1.5.

3. List all Bitwise operator. [2 M]

Ans. Refer to Section 1.5.

Winter 2015

1. What is keyword? Explain with example. [2 M]

Ans. Refer to Section 1.4.3.

2. List all relational operators in 'C'. [2 M]

Ans. Refer to Section 1.5.

3. What are applications of 'C'? [5 M]

Ans. Refer to Section 1.1.

Summer 2016

1. What is identifier? Explain with example. [2
Ans. Refer to Section 1.4.4.
2. Define operator. List any four types of operators. [2
Ans. Refer to Section 1.5.
3. Explain structure of 'C' program with example. [5
Ans. Refer to Section 1.3.

Winter 2016

1. List primary data types in C language. [2
Ans. Refer to Section 1.4.7.
2. Define operator. List and explain any four categories of operators in C. [5
Ans. Refer to Section 1.5.

Summer 2017

1. 'C' is middle level language. Comment. [2
Ans. Refer to Section 1.1.
2. What is escape sequence? [2
Ans. Refer to Section 1.4.5.

Summer 2018

1. What are various data types used in C? [2
Ans. Refer to Section 1.4.7.
2. Define keyword. [2
Ans. Refer to Section 1.4.3.
3. What do you mean by scope of a variable? [2
Ans. Refer to Section 1.4.6.

2...

Managing I/O Operations

Learning Objectives...

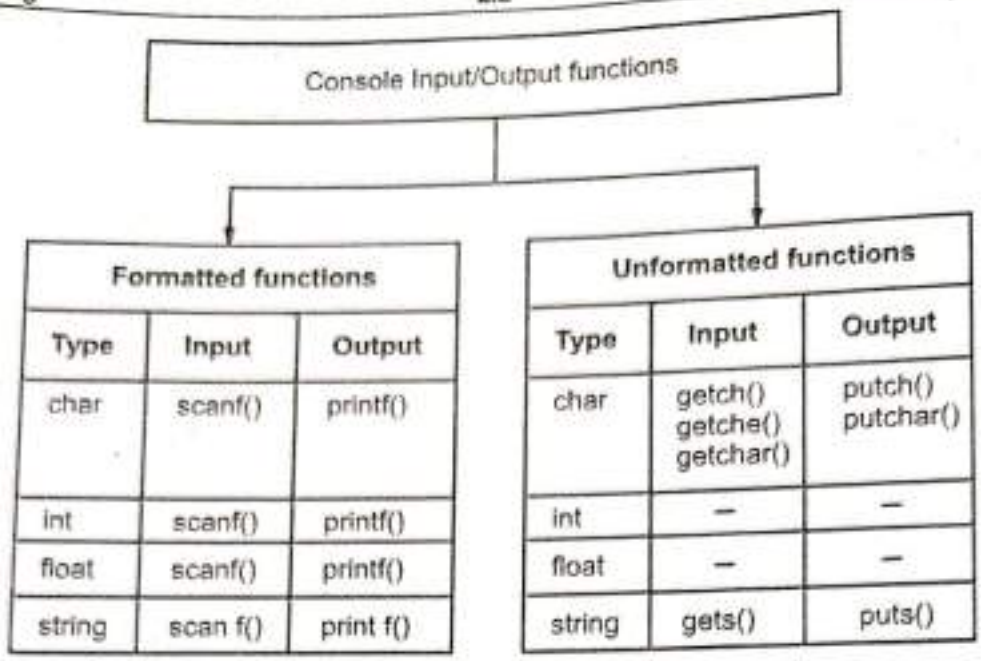
- To study functionality of programming.
- To interpret the importance of C functions.
- To understand the Console based I/O and Related Built-in I/O functions.

2.1 INTRODUCTION

- In C, Input and Output operations are performed using library functions – without these functions we cannot interact with the compiler.
- We can perform input functions using input devices i.e. keyboard and output functions using output devices i.e. screen or printer and having interaction with compiler through these devices.
- There are two Console and File Input/Output (I/O) functions. Here, we refers to the console I/O functions as performing input from the keyboard and output to the screen.
- To use all these I/O functions we have to include as important header file **stdio.h** using statement, `#include<stdio.h>`. This statement is included before every program.
- There are numerous library functions available for I/O. These can be classified into two broad categories:
  1. **Console I/O functions:** Functions to receive input from keyboard and write output to VDU (Visual Display Unit).
  2. **File I/O functions:** Functions to perform I/O operations on a floppy disk or hard disk.

2.2 CONSOLE BASED I/O AND RELATED BUILT-IN I/O FUNCTIONS

- The screen and keyboard together are called a Console. Console I/O functions can be further classified into two categories – Formatted and Unformatted Console I/O Functions.
- The functions available under each of these two categories are shown in Fig. 2.1.



Formatted functions		
Type	Input	Output
char	scanf()	printf()
int	scanf()	printf()
float	scanf()	printf()
string	scan f()	print f()

Unformatted functions		
Type	Input	Output
char	getch() getche() getchar()	putch() putchar()
int	–	–
float	–	–
string	gets()	puts()

Fig. 2.1: Function Categories

2.2.1 printf() Function

[S-16]

- printf() function is used to display output of a program on screen.
- printf() is one of the library function.

  Syntax: `printf("information string", arg1, arg2, ... , argn)`

  where, information string refer to a string that containing formatting information and arg1, arg2, ...argn are arguments that represent the individual output data items.
- Escape Sequence is a pair of character. The first letter is a slash followed by a character. It actually represents only one character.
- The Conversion specification describes the printf() function that it could print some value at that location in the text.
- Format (Access) Specifiers are:

%d	Data item is displayed as a signed decimal integer.
%i	Data item is displayed as a single decimal integer.
%f	Data item is displayed as a floating-point value without an exponent.
%c	Data item is displayed as a single character.
%e	Data item is displayed as a floating-point value with an exponent.
%g	Data item is displayed as a floating-point value using either e-type or f-type conversion depending on value.
%o	Data item is displayed as an octal integer, without a leading zero.
%s	Data item is displayed as string.
%u	Data item is displayed as an unsigned decimal integer.
%x	Data item is displayed as a hexadecimal integer, without a leading 0x.

Program 2.1: Program for C printf().

```
#include <stdio.h>
int main()
{
    char ch;
    char str[100];
    printf("Enter any character \n");
    scanf("%c", &ch);
    printf("Entered character is %c \n", ch);
    printf("Enter any string ( upto 100 character ) \n");
    scanf("%s", &str);
    printf("Entered string is %s \n", str);
    return 0;
}
```

Output:

```
Enter any character
a
Entered character is a
Enter any string ( upto 100 character )
hai
Entered string is hai
```

- The format specifier %c is used in scanf statement so that the value entered is received as an character and %s for string.

2.2.2 scanf() Function

[W-15, 16, S-17]

- To read data from the user we use scanf().
- The scanf( ) is used to read all the built-in data types and automatically convert numbers into the proper internal format.
- scanf() is the general purpose console input routine.

  Syntax: `scanf("information string", arg1, arg2, ..... argn);`

  where, information string consists of, format specifiers, white space characters (tab, newline, blank), non-white-space characters which causes scanf to discard the matching character etc.

  Example: `scanf("%d", &a);`
- Here, we consider a is integer, so we use %d and a value will be read from keyboard and this will automatically given to a, Also &a reserves the memory location for the value.

Program 2.2: Program to find average of three subjects of student.

```
#include<stdio.h>
void main( )
{
    char name[30] ;
    int a, b, c ;
    float result ;
    printf("\n please enter name:") ;
    scanf (" %s", name) ;
    printf("\n please enter mark 1:") ;
    scanf (" %d", &a);
    printf(" \n please enter mark 2: ") ;
    scanf ("%d", &b);
    printf ("\n Please enter mark 3: ");
    scanf("%d", &c);
    result = (a + b + c)/3 ;
    printf("\n average = %f", result) ;
    getch( ) ;
}
```

Output:

```
Please enter name          :   Omkar
Please enter mark 1        :   80
Please enter mark 2        :   75
Please enter mark 3        :   60
average = 71.0000
```

2.2.3 getchar() Function

- getchar() is used to get or read the input (i.e a single character) at run time.

Syntax: `int getchar(void);`

Example:

```
void main()
{
char ch;
ch = getchar();
printf("Input Char Is:%c",ch);
}
```

Program Explanation:

- Here, declare the variable **ch** as char data type, and then get a value through getcha library function and store it in the variable ch. And then print the value variable ch.
- During the program execution, a single character is get or read through the **getcha** The given value is displayed on the screen and the compiler wait for anot character to be typed. If you press the enter key/any other characters and then o the given character is printed through the **printf** function.

2.2.4 getch() Function

- getch() is used to get a character from console but does not echo to the screen.
  Syntax: `int getch(void);`
- getch() reads a single character directly from the keyboard, without echoing to the screen.

```
void main()
{
   char ch;
   ch = getch();
   printf("Input Char Is:%c",ch);
}
```

Program Explanation:

- Here, declare the variable **ch** as char data type, and then get a value through getch() library function and store it in the variable ch. And then, print the value of variable ch.
- During the program execution, a single character is get or read through the **getch()**. The given value is not displayed on the screen and the compiler does not wait for another character to be typed. And then, the given character is printed through the **printf** function.

2.3 FORMATTED INPUT AND OUTPUT FUNCTIONS [W-15, S-16]

1. getche() Function:

- getche() is used to get a character from console, and echoes to the screen.
  Syntax: `int getche(void);`
- getche reads a single character from the keyboard and echoes it to the current text window, using direct video or BIOS.

Example:

```
void main()
{
   char ch;
   ch = getche();
   printf("Input Char Is:%c",ch);
}
```

Program Explanation:

- Here, declare the variable **ch** as char data type, and then get a value through **getche()** library function and store it in the variable **ch**. And then, print the value of variable **ch**.
- During the program execution, a single character is get or read through the **getche()**. The given value is displayed on the screen and the compiler does not wait for another character to be typed. Then, afterwards the character is printed through the **printf** function.

2. sprintf( ) Function:

- The sprintf function converts, formats, and stores its value parameters under contr of the format parameter.

  Syntax: `str=sprintf(format,value_1,..,value_n)`

- Parameters:
  - **format:** a Scilab string. Specifies a character string combining literal characte with conversion specifications.
  - **value , ... , value_n:** Specifies the data to be converted according to the forma parameter.
  - **str:** column vector of character strings.

  The format parameter is a character string that contains two types of objects:
  - **Literal Characters:** which are copied to the output stream.
  - **Conversion Specifications**: each of which causes zero or more items to be fetche from the value parameter list.

3. sscanf( ) Function:

- sscanf() converts formatted input given by a string. The sscanf functions interpr character string according to a format, and returns the converted results. The forma parameter contains conversion specifications used to interpret the input.
- The format parameter can contain white-space characters (blanks, tabs, newline, formfeed).
- Unless there is a match in the control string, trailing white space (including a newlin character) is not read.
  1. Any character except % (percent sign), which must match the next character o the input stream.
  2. A conversion specification that directs the conversion of the next input field.

  Syntax: `[v_1,...v_n]=sscanf (string,format)`

- Parameters:
  - **format:** Specifies the format conversion.
  - **string:** Specifies input to be read.

4. String Input and Output functions:

- The functions gets() and puts() used for string reading (input) and string writing (output) purpose from standard library functions.

(i) gets() Function: [W-16

- This function reads a string of characters entered at the keyboard. You can type the characters at the keyboard until a carriage return or ENTER is pressed.
- The end of string is denoted by null character (\0). Whitespaces (tab and spaces) are allows in a string.

  Syntax: `gets(string_name)`

Program 2.3: Program for gets().

```
#include<stdio.h>
void main()
{
char str[80];
gets(str);
}
```

- In the above program, we have demonstrated the use of gets(). The scanning or reading the string from keyboard continues till the <Enter> key is pressed.

(ii) puts() Function:

- This function is used to print a string on the screen. It is equivalent to printf() function, but in printf statement we can display messages along with printing at the same time.
- The puts() function can only output a string of characters. It cannot output numbers or do format conversions. Therefore, puts() requires less space than printf(). It runs faster than printf().

Syntax: `puts(string_name)`

Program 2.4: Program for puts().

```
#include<stdio.h>
void main()
{
   char str[80];
   gets(str);
   puts(str);
   printf("The line is: %s", str);
}
```

Output:

```
This is my string data
This is my string data
The line is: This is my string data (writing by printf)
```

- So we can see that the main use of gets and puts is to transfer the line of text to and from the computer.

5. putchar() Function: [S-17]

- The putchar() function writes a character to the screen at the current cursor position. We can return a single character and print it on the screen.

Syntax: `putchar(char_variable);`

- The value of char_variable is print on the screen.

Program 2.5: Program to calculate area of circle (formula: π * r * r).

```
#include<stdio.h>
#include<conio.h>
void main()
{
   float radius,area;
```

```
clrscr(); // Clear Screen
printf("\n Enter the radius of Circle: ");
scanf("%f",&radius);
area = 3.14 * radius * radius;
printf("\n Area of Circle: %f",area);
getch();
}
```

Output:

```
Enter the radius of Circle: 2.0
Area of Circle: 12.56
```

Program 2.6: Program to calculate area of square, (Formula area: side * side).

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int side,area;
   clrscr(); // Clear Screen
   printf("\n Enter the Length of Side: ");
   scanf("%d",&side);
   area = side * side ;
   printf("\n Area of Square: %d",area);
   getch();
}
```

Output:

```
Enter the Length of Side: 4
Area of Square: 16
```

Program 2.7: Program to calculate area of rectangle (Formula: area = l * b).

```
#include<stdio.h>
#include<conio.h>
void main()
{
int length,breadth,side, area;
clrscr(); // Clear Screen
printf("\n Enter the Length of Rectangle: ");
scanf("%d",&length);
printf("\n Enter the Breadth of Rectangle: ");
scanf("%d",&breadth);
area = length * breadth;
printf("\n Area of Rectangle: %d",area);
getch();
}
```

Output:

```
Enter the Length of Rectangle: 5
Enter the Breadth of Rectangle: 6
Area of Rectangle: 30
```

Program 2.8: Program to convert temperature fahrenheit into celsius. **[S-16]**

```
#include<stdio.h>
#include<conio.h>
void main()
{
 float Fahrenheit, Celsius;
 clrscr();
 printf("Enter Temperature in Fahrenheit \n");
 scanf("%f",&Fahrenheit);
 Celsius = 5.0/9.0 * (Fahrenheit-32);
 printf("\n Temperature in Fahrenheit = %f", Fahrenheit);
 printf("\n Temperature in Celsius = %f", Celsius);
 getch();
}
```

Output:

```
Enter Temperature in Fahrenheit
15.2
Temperature in Fahrenheit = 15.2000
Temperature in Celsius = -9.3333
```

Summary

- To use all these I/O functions we have to include as important header file 'stdio.h' using statement, #include<stdio.h>. This statement is included before every program.
- The screen and keyboard together are called a **console**. Console I/O functions can be further classified into two categories – Formatted and Unformatted Console I/O Functions.
- printf() function is used to display output of a program on screen.
- The scanf( ) is used to read all the built-in data types and automatically convert numbers into the proper internal format.
- getchar() is used to get or read the input (i.e a single character) at run time.
- getch() reads a single character directly from the keyboard, without echoing to the screen.
- getche() is used to get a character from console, and echoes to the screen.

- The sprintf function converts, formats, and stores its value parameters, under contr of the format parameter.
- sscanf() converts formatted input given by a string. The sscanf functions interpr character string according to a format, and returns the converted results.
- gets() function reads a string of characters entered at the keyboard. You can type t characters at the keyboard until a carriage return or ENTER is pressed.
- puts() function is used to print a string on the screen. It is equivalent to print function, but in printf statement we can display messages along with printing at t same time.

Check Your Understanding

1. putchar(c) function/macro always outputs character c to the _____.
   (a) screen
   (b) standard output
   (c) depends on the compiler
   (d) depends on the standard
2. What are the Properties of first argument of a printf() functions?
   (a) It is defined by user.
   (b) It keeps the record of the types of arguments that will follow.
   (c) There may no be first argument.
   (d) None of the mentioned.
3. What is the purpose of sprintf?
   (a) It prints the data into stdout.
   (b) It writes the formatted data into a string.
   (c) It writes the formatted data into a file.
   (d) None of the mentioned.
4. The syntax to print a % using printf statement can be done by _____.
   (a) %
   (b) \%
   (c) '%'
   (d) %%
5. Which among the following is the odd one out?
   (a) printf
   (b) fprintf
   (c) putchar
   (d) scanf
6. For a typical program, the input is taken using _____.
   (a) scanf
   (b) Files
   (c) Command-line
   (d) All of the mentioned

Answers

1. (b)	2. (b)	3. (b)	4. (d)	5. (d)	6. (d)

Trace The Output

1.
```
void main()
{
    int a=100*10,b;
    b=a/2;
    printf("\n%d\n%d",a,b);
}
```

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

3.
```
void main()
{
    int x=8,y;
    y=x--;
    y-=--x;
    printf("%d %d",x,y);
}
```

4.
```
void main()
{
    int Float=2,pi=3.14;
    printf("%f%f",Float,pi);
}
```

5.
```
void main()
{
    char c='z',ch;
    c=c+'a'-'A';
    ch=c-'a'+'A';
    printf("%c",ch);
}
```

Practice Questions

Q.1 Write the answer of following questions in brief:

1. What is meant by built-in I/O functions?
2. Which header file is used for input-output operations?
3. Explain the purpose of the function sprintf().
4. Which I/O function used for string?
5. Which format access specifier used for integer.

Q.2 Write the answer of following questions:

1. State two functions for reading the characteristics from the user.

2. What is the difference between puts() and putchar().
3. What is a purpose of scanf() function? State any four format specifies used with scanf().
4. What is the purpose of printf() function? State any four format specifiers used in printf( ) statement with an example.
5. Write a program to find the area of a rectangle.
6. Write a program to find volume of a cylinder.

Q.3 Define terms:
1. printf(), 2. scanf(), 3. puts(), 4. gets(), 5. getch(), 6. getche(), 7. sprintf(), 8. sscanf().

Previous Exam Questions

Winter 2014

1. Write a 'C' program to convert temperature from celcius to faharanite. [5 M]

Ans. Refer to Program 2.8.

Winter 2015

1. Give syntax of scanf statement with example. [2 M]

Ans. Refer to Section 2.2.2.

2. Usage of getchar() and gets() with example. [2 M]

Ans. Refer to Sections 2.2.3 and 2.3.

Summer 2016

1. Give syntax of printf statement with example. [2 M]

Ans. Refer to Section 2.2.1.

2. What is the usage of putchar() and puts(). [2 M]

Ans. Refer to Section 2.3.

3. Write a 'C' program to convert temperature from Celsius to Fahrenheit. [5 M]

Ans. Refer to Program 2.8.

Winter 2016

1. What is the difference between scanf() and gets()? [2 M]

Ans. Refer to Sections 2.2.2 and 2.3.

Summer 2017

1. Give syntax and example of (i) scanf(), (ii) putchar(). [2 M]

Ans. (i) **scanf( ):** Refer to Section 2.2.2.
(ii) **putchar():** Refer to Section 2.3.

3... Decision Making and Looping

Learning Objectives...

- To study occurrence of conditions.
- To execute certain instructions statements in programs.
- To study various looping and decision making statements and its applications.
- To understand importance of decision making and looping statements.

3.1 INTRODUCTION [S-18]

- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed, if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- Fig. 3.1 shows the general form of a typical decision making structure that found in most of the programming languages.
- C programming language assumes any non-zero and non-null values as true and if it is either zero or null then it is assumed as false value.

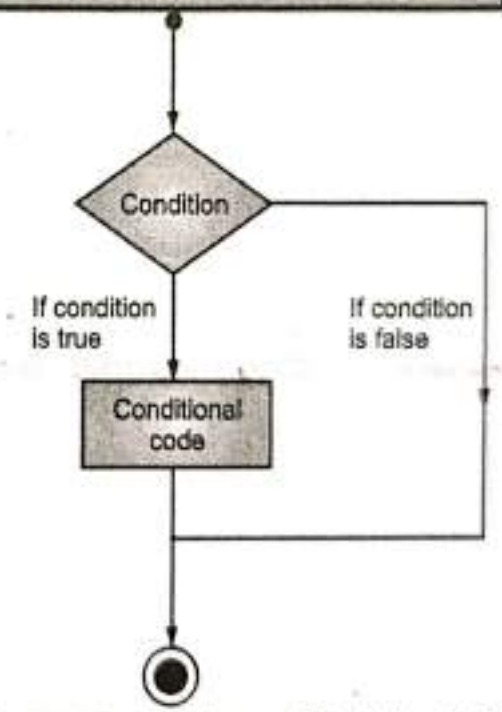


Fig. 3.1: General form of Decision-Making Structure

- C programming language provides following types of decision making statements.

Statement	Description
1. if statement	An if statement consists of a boolean expression followed by one or more statements.

contd. ...

2. if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
3. Nested If Statements	You can use spacing if statement inside another if or else if statements.
4. switch statement	A switch statement allows a variable to be tested for equality against a list of values.
5. Nested Switch Statements	You can use one switch statement inside another switch statements.

- There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.
- A loop statement allows us to execute a statement or group of statements multiple times. Fig. 3.2 shows the general form of a loop statement in most of the programming languages.

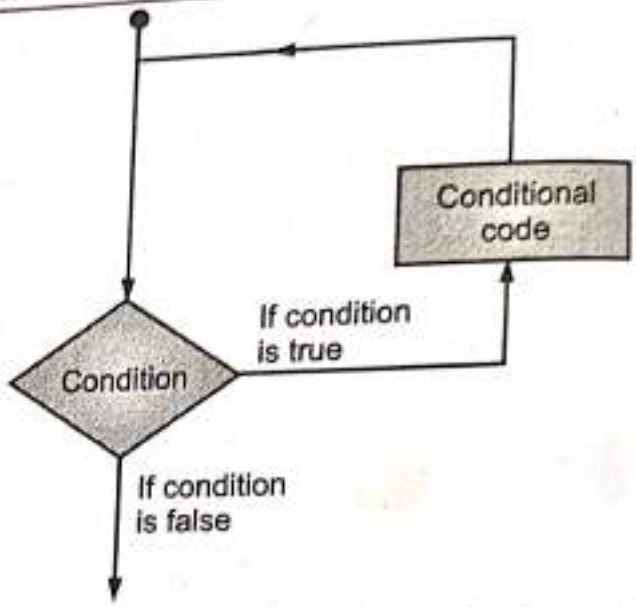


Fig. 3.2: General form of a Loop Statement

- C programming language provides following types of loop to handle looping requirements.

Type of Loop	Description
1. while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2. for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3. do...while loop	Like a while statement, except that it tests the condition at the end of the loop body.

- C provides following types of statements:
  1. **Expression Statement:** Number of expression statements are assignments or function calls.
  2. **Compound Statement:** More than two statements grouped together in { } forms a compound statement.
  3. **Selection Statement:** These statements involve condition checking and choose one of several flows of controls. if, if ... else, switch are the examples of selection statement.

4. **Null Statement:** These statements does not perform any action. A semicolon (;) on the line of code is a Null statement.
5. **Iteration Statement:** These statement specify looping, where a statement has to be repeatedly executed a specific number of times as the test expression is satisfied. For, do while and while loop are the examples of iteration statement.
6. **Jump Statement:** These statements transfer control unconditionality. goto, break, continue are the examples of Jump statements.

3.2 DECISION MAKING STRUCTURE

- C program executes program sequentially. Sometimes, a program requires checking of certain conditions in program execution.
- C provides various key condition statements to check condition and execute statements according conditional criteria.
- These statements are called as 'Selection' or 'Conditional Statements.'
- Fig. 3.3 shows Decision Making Statement.

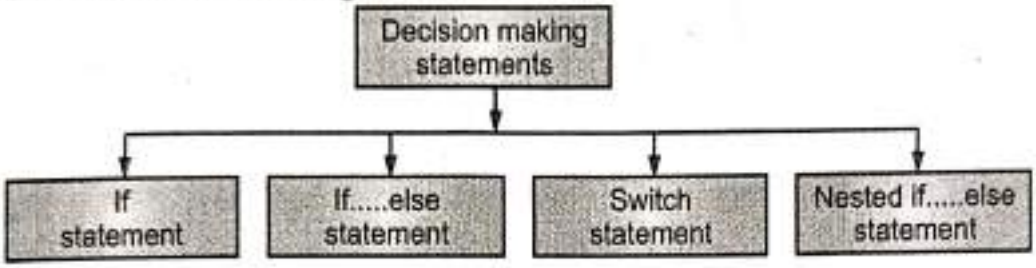


Fig. 3.3: Types of decision making statements

3.2.1 if Statement

- This is a conditional statement used in C to check condition or to control the flow of execution of statements.
- This is also called as 'decision making statement or control statement.'
- The execution of a whole program is done in one direction only.

Syntax:

```
if(condition)
{
   statements;
}
```

- In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and executes the block of statements associated with it. If it returns false, then program skips the statements in braces. If there are more than 1 (one) statements in if statement then use { } braces else it is not necessary to use.
- Fig. 3.4 shows flowchart of if statement.

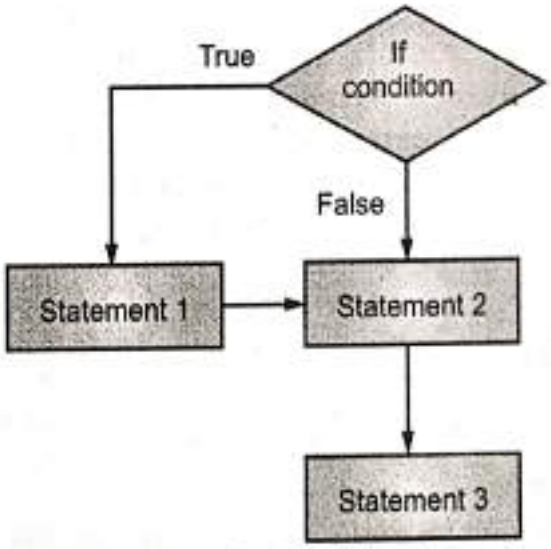


Fig. 3.4: Flowchart of if statement

Program 3.1: Program for if statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num;
    clrscr();
    printf("\n Enter a number less than 10:");
    scanf("%d", &num);
    if(num <= 10)
    printf("\n The number is less than 10");
    getch();
}
```

Output:

First run:

```
Enter a number less than 10 : 8
The number is less than 10
```

Second run:

```
Enter a number less than 10 : 14
```

- After second run there will be no output because the condition given in the statement is false. The 'if' statements are only executed when the condition is true.

Program 3.2: C program to print the number entered by user only if the number entered is negative.

```
#include <stdio.h>
int main()
{
    int num;
    printf("Enter a number to check.\n");
    scanf("%d",&num);
    if(num<0) /* checking whether number is less than 0 or not. */
    printf("Number=%d\n",num);
    /*If test condition is true, statement above will be executed, otherwis
    it will not be executed */
    printf("The if statement in C programming is easy.");
    return 0;
}
```

Output 1:

```
Enter a number to check.
-2
Number=-2
The if statement in C programming is easy.
```

- When user enters – 2 then, the test expression (num<0) becomes true. Hence, Number = –2 is displayed in the screen.

Output 2:

```
Enter a number to check.
5
The if statement in C programming is easy.
```

3.2.2 if-else Statement [S-17]

- This is also one of the most useful conditional statement used in C to check conditions.
- The if statement will execute the statement if the expression is true otherwise it will be skipped.
- The if-else statement allows us to select one of the two variable options depending upon outcome of test condition.

Syntax:

```
if(condition)
{
   true statements;
}
else
{
   false statements;
}
```

- In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and executes the block of statements associated with it. If it returns false, then it executes the else part of a program.
- Fig. 3.5 shows flowchart for if...else statement.

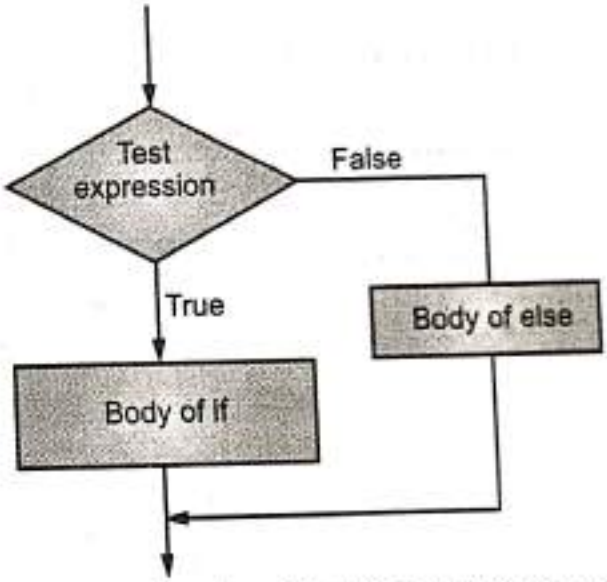


Fig. 3.5: Flowchart for if...else statement

Program 3.3: Program to check whether input alphabet is a vowel or not.

```
#include <stdio.h>
int main()
{
   char ch;
   printf("Enter a character\n");
   scanf("%c", &ch);
   if (ch == 'a' || ch == 'A' || ch == 'e' || ch == 'E' || ch == 'i' || ch
   == 'I' || ch =='o' || ch=='O' || ch == 'u' || ch == 'U')
   printf("%c is a vowel.\n", ch);
   else
   printf("%c is not a vowel.\n", ch);
   return 0;
}
```

Output 1:

```
Enter a character
i
i is a vowel.
```

Output 2:

```
Enter a character
x
x is not a vowel.
```

Program 3.4: Program to add two complex numbers.

```
#include <stdio.h>
/*We use structure for this program. For structure details see Chapter 6*/
struct complex
{
   int real, img;
};
int main()
{
   struct complex a, b, c;
   printf("Enter a and b where a + ib is the first complex number.\n");
   printf("a = ");
   scanf("%d", &a.real);
   printf("b = ");
   scanf("%d", &a.img);
   printf("Enter c and d where c + id is the second complex number.\n");
   printf("c = ");
   scanf("%d", &b.real);
   printf("d = ");
   scanf("%d", &b.img);
   c.real = a.real + b.real;
   c.img = a.img + b.img;
   if ( c.img >= 0 )
   printf("Sum of two complex numbers = %d + %di\n",c.real,c.img);
   else
   printf("Sum of two complex numbers = %d %di\n",c.real,c.img);
   return 0;
}
```

Output:

```
Enter a and b where a + ib is the first complex number.
a = 3
b = 4
Enter c and d where c + id is the second complex number.
c = 2
d = 1
Sum of two complex numbers = 5 + 5i
```

Nested if Statement:

- A Nested if statement is simply an if statement within an if statement.
- The Nested if statement enables us to specify multiple actions in a single instruction.
- The **syntax for a Nested if** statement is as follows:

```
if( boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2)
   {
      /* Executes when the boolean expression 2 is true */
   }
}
```

Program 3.5: Program to demonstrate nested if-else statement.

```
#include<stdio.h>
#include<conio.h>
int main()
{
   int num;
   printf("Enter a number\n");
   scanf("%d", &num);
   if(num < 500)
   {
      printf("First Condition is true\n");
      if(num > 100)
      {
         printf("First and Second conditions are true\n");
       }
   }
   printf("This will print always\n");
   getch();
   return(0);
}
```

Output

```
Enter a number
80
First Condition is true
This will print always
Enter a number
200
First Condition is true
First and Second conditions are true
This will print always
Enter â number
800
This will print always
```

Program 3.6: Program for GCD and LCM of two integers.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1, num2, gcd, lcm, remainder, numerator, denominator;
    clrscr();
    printf("Enter two numbers\n");
    scanf("%d %d", &num1,&num2);
    if (num1 > num2)
    {
        numerator = num1;
        denominator = num2;
    }
    else
    {
        numerator = num2;
        denominator = num1;
    }
    remainder = num1 % num2;
    while(remainder !=0)
    {
        numerator = denominator;
        denominator = remainder;
        remainder = numerator % denominator;
    }
    gcd = denominator;
    lcm = num1 * num2 / gcd;
    printf("GCD of %d and %d = %d \n", num1,num2,gcd);
    printf("LCM of %d and %d = %d \n", num1,num2,lcm);
    }     /* End of main() */
```

Output:

```
Enter two numbers
5
15
GCD of 5 and 15 = 5
LCM of 5 and 15 = 15
```

3.2.3 Nested if-else Statement

- It is a conditional statement which is used when we want to check more than conditions at a time in a same program.
- The conditions are executed from top to bottom checking each condition whether meets the conditional criteria or not.

Syntax:

```
if(condition)
{
    if(condition)
    {
        statements;
    }
    else
    {
        statements;
    }
}
else
{
    statements;
}
```

- In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and again checks the next condition. If it is true then it executes the block of statements associated with it else executes else part.

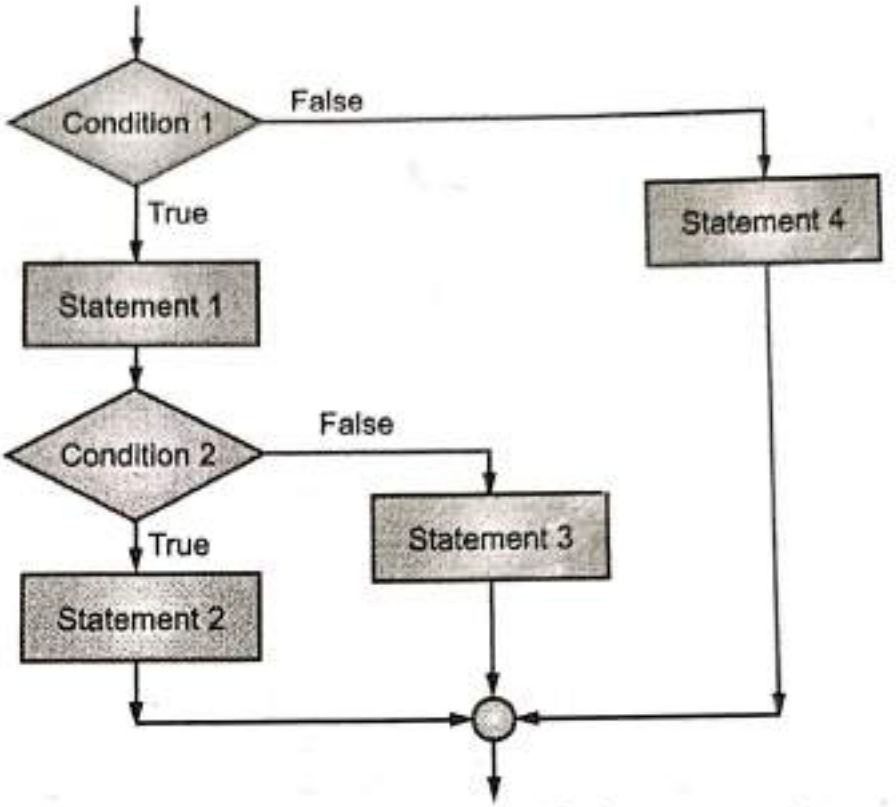


Fig. 3.6: Flowchart for nested if-else statement

- Fig. 3.6 shows flowchart for nested if-else statement.

Program 3.7: Program to demonstrate nested if-else statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int no;
   clrscr();
   printf("\n Enter Number:");
   scanf("%d",&no);
   if(no>0)
   {
      printf("\n\n Number is greater than 0!");
   }
   else
   {
      if(no==0)
      {
```

```
            printf("\n\n It is 0!");
            }
            else
            {
            printf("Number is less than 0 !");
            }
        }
        getch();
    }
```

Output 1:

```
Enter Number: 0
It is 0!
```

Output 2:

```
Enter Number: 7
Number is greater than 0!
```

else-if Ladder:

- In C programming language the else if ladder is a way of putting multiple ifs together when multipath decisions are involved. It is a one of the types of decision making branching statements.
- A multipath decision is a chain of if's in which the statement associated with each else is an if statement.
- The **syntax** of else if ladder is as follows:

```
if (condition_1)
{
   statement_1;
}
else if (condition_2)
{
   statement_2;
}
else if (condition_n)
{
   statement_n;
}
else
{
   default statment;
}
statement-x;
```

- Above construct is known as the 'else-if' ladder. The conditions are evaluated from the top of the ladder to downwards.

- As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the default statement will be executed.
- Fig. 3.7 shows flowchart for if-else ladder.

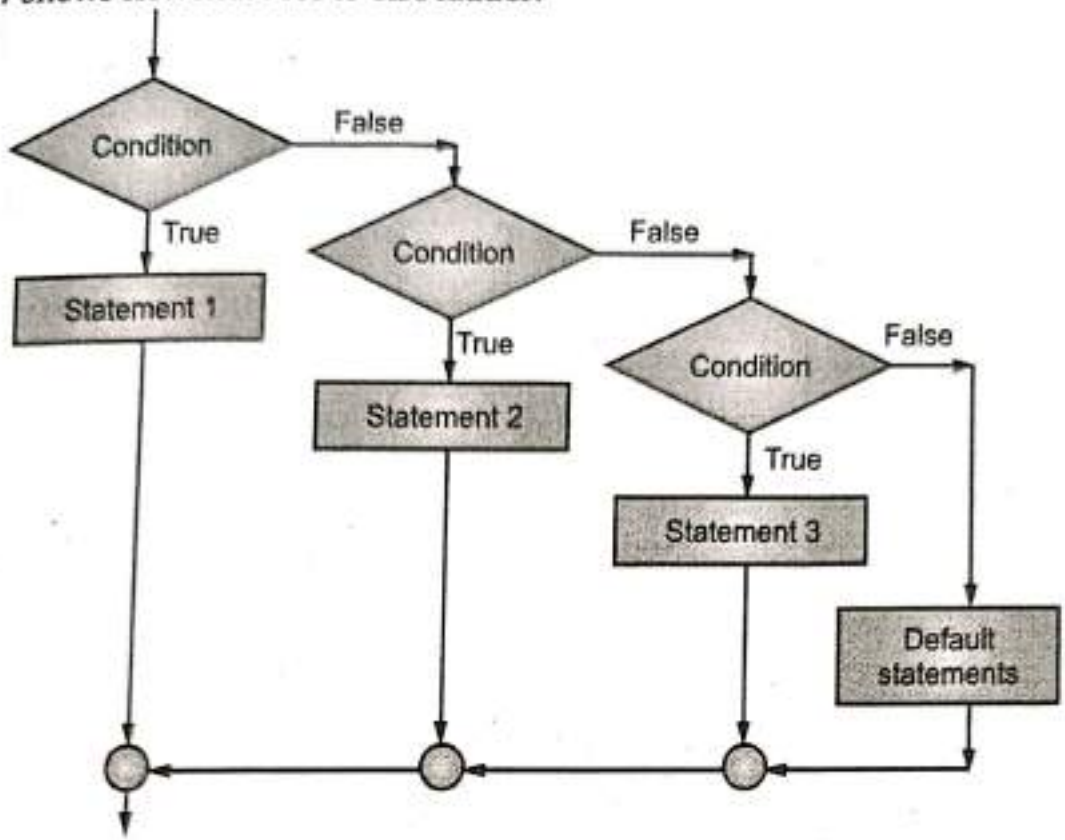


Fig. 3.7: Flowchart for if-else ladder

- Below is the sample C program of the if – else ladder statement in which the color is to beselected by using the if – else ladder.

Program 3.8: Program to demonstrate else-if ladder.

```
#include<stdio.h>
#include<string.h>
void main()
{
   int n;
   printf(" Enter 1 to 4 to select random color");
   scanf("%d",&n);
   if(n==1)
   {
      printf("You selected Red color");
   }
   else if(n==2)
   {
      printf("You selected Green color");
   }
```

```
    else if(n==3)
    {
        printf("You selected yellow color");
    }
    else if(n==4)
    {
        printf("You selected Blue color");
    }
    else
    {
        printf("No color selected");
    }
    getch();
}
```

Output:

```
Enter 1 to 4 to select random color 3
You selected yellow color
```

3.2.4 switch Statement [S-17]

- Switch statement is a multiple or multiway branching decision making statement.
- When we use nested if-else statement to check more than one conditions then the complexity of a program increases in case of a lot of conditions. Thus, the program is difficult to read and maintain. So to overcome this problem, C provides 'switch statement'.
- Switch statememt checks the value of a expression against a case values, if condition matches the case values then the control is transferred to that point.

Syntax:

```
switch(expression)
{
    case expr1:
        statements;
        break;
    case expr2:
        statements;
        break;
    case exprn:
        statements;
        break;
    default:
        statements;
}
```

- In above syntax, switch, case, break are keywords. expr1, expr2 are known as 'case Labels.' Statements inside case expression need not to be closed in braces. Break statement causes an exit from switch statement. Default case is optional case. When neither any match found, it executes.
- Fig. 3.8 shows flowchart of switch statement.

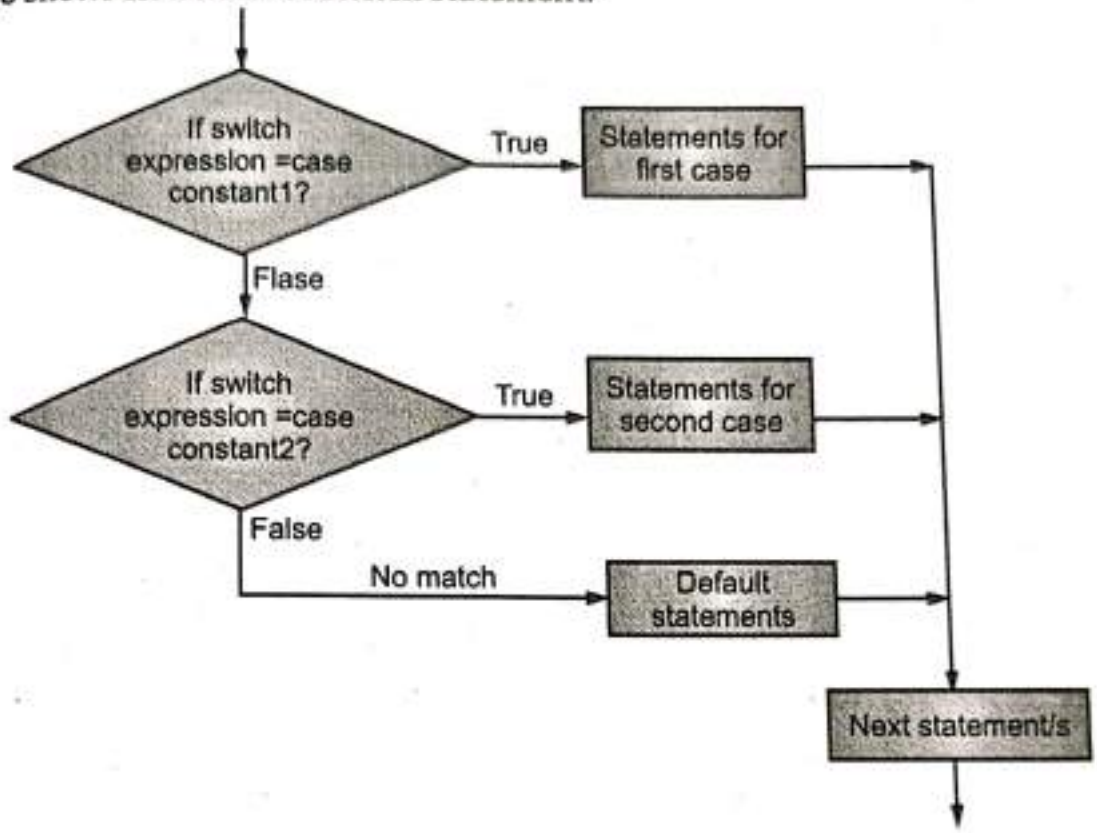


Fig. 3.8: Flowchart of switch...case statement

- Rules for declaring switch ... case statement:
  1. Case labels must end with (:) colon.
  2. The case label should be integer or character constant.
  3. Each compound statement of a switch case should contain break statement to exit from case.

Program 3.9: Write a program to demonstrate switch statement.

```
/*switch statement demonstration*/
#include<stdio.h>
#include<conio.h>
void main()
 {
   int color = 1;
   printf("Please choose a color \n");
   printf("1: red\t\t2: green\t\t3: blue:\n");
   scanf("%d", &color);
   switch (color)
   {
      case 1:
      printf("you choose red color\n");
         break;
```

```
        case 2:
        printf("you choose green color\n");
           break;
        case 3:
        printf("you choose blue color\n");
           break;
        default:
        printf("you did not choose any color\n");
     }
     getch();
  }
```

Output:

```
Please choose a color
1: red    2: green          3: blue:
2
you choose green color
```

Program 3.10: Program to check entered character is vowel or not.

```
/*check vowel or not using switch*/
#include<stdio.h>
#include<conio.h>
void main()
{
  char ch;
  clrscr();
  printf("Enter a character\n");
  scanf("%c", &ch);
  switch(ch)
  {
    case 'a':
    case 'A':
    case 'e':
    case 'E':
    case 'i':
    case 'I':
    case 'o':
    case 'O':
    case 'u':
    case 'U':
      printf("%c is a vowel.\n", ch);
      break;
    default:
      printf("%c is not a vowel.\n", ch);
  }
getch();
}
```

Output:

```
Enter a character
R
R is not a vowel.
```

Advantages of switch ... case statement:

1. Complexity of a program is minimized.
2. Easy and simple to use and understand.
3. Easy to find out errors.

Nested Switch Statement:

- Switch statement in another switch statement is called Nested switch.
- The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
   case 'A':
      printf("This A is part of outer switch" );
      switch(ch2) {
   case 'A':
      printf("This A is part of inner switch" );
      break;
   case 'B': /* case code */
}
break;
case 'B': /* case code */
}
```

Program 3.11: Program for nested switch statement.

```
#include <stdio.h>
int main ()
{
 /* local variable definition */
 int a = 100;
 int b = 200;
 switch(a)
{
 case 100:
 printf("This is part of outer switch\n", a );
 switch(b)
{
 case 200:
 printf("This is part of inner switch\n", a );
 }
 }
 printf("Exact value of a is: %d\n", a );
 printf("Exact value of b is: %d\n", b );
 return 0;
}
```

Output:

```
This is part of outer switch
```

Advantages of switch ... case statement:

1. Complexity of a program is minimized.
2. Easy and simple to use and understand.
3. Easy to find out errors.

Nested Switch Statement:

- Switch statement in another switch statement is called Nested switch.
- The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
   case 'A':
      printf("This A is part of outer switch" );
      switch(ch2) {
   case 'A':
      printf("This A is part of inner switch" );
      break;
   case 'B': /* case code */

}
break;
case 'B': /* case code */
}
```

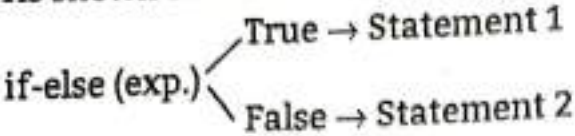
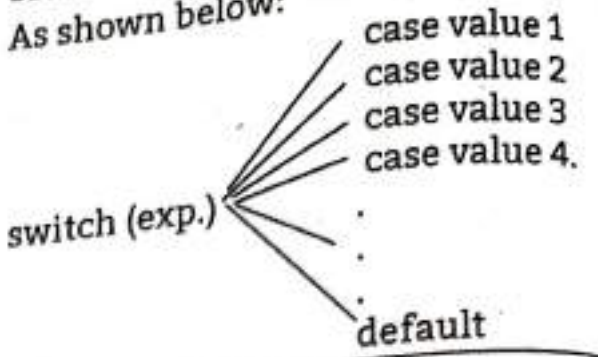
Program 3.11: Program for nested switch statement.

```
#include <stdio.h>
int main ()
{
 /* local variable definition */
 int a = 100;
 int b = 200;
 switch(a)
{
 case 100:
 printf("This is part of outer switch\n", a );
 switch(b)
{
 case 200:
 printf("This is part of inner switch\n", a );
 }
 }
 printf("Exact value of a is: %d\n", a );
 printf("Exact value of b is: %d\n", b );
 return 0;
}
```

Output:

```
This is part of outer switch
This is part of inner switch
Exact value of a is: 100
Exact value of b is: 200
```

Difference between if-else and switch statements:

if-else Statement	switch Statement
1. This statement allows only two way branching from a single expression. As shown below: 	1. This statement allows multiway branching from as single expression. As shown below: 
2. Putting { and } braces is essential.	2. switch statement belonging to a case need not be need { and } braces.
3. Tracing of errors and debugging is expensive and difficult.	3. Tracing of errors and debugging is easy.
4. This statement is difficult to write.	4. This statement is easier to write.
5. Using this statement code/program become complex.	5. Code become simple and easy.

6. **Syntax:**
`if condition`
`{`
`statements …`
`}`
`else`
`{`
`statements …`
`}`
	6. **Syntax:**
	`switch (variable)`
	`{`
	`case value 1:`
	`break;`
	`case value 2:`
	`break;`
	`:`
	`default:`
	`}`

3.2.5 Conditional Operator

- This is the only ternary operator in C language. It is denoted by ?: symbol.
- Conditional operator (?:) has the following general form:

```
Exp1? Exp2: Exp3;
```

Where, Exp1, Exp2, and Exp3 are expressions.

- The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, the Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false then Exp3 is evaluated and its value becomes the value of the expression.

Program 3.12: Check whether number is Odd or Even using conditional operators.

```
#include<stdio.h>
int main()
{
   int num, flag;
   printf("Enter the Number: ");
```

```
    scanf("%d",&num);
       flag = ((num%2==0)?0:1);
    if(flag==0)
     printf("\nEven");
    else
     printf("\nOdd");
     return(0);
}
```

Output 1:

```
Enter the Number: 4
Even
```

Output 2:

```
Enter the Number: 3
Odd
```

3.3 LOOP CONTROL STRUCTURES [S-16]

- Iterative statements are used to run a particular block statements repeatedly or in other words 'from a loop'.
- A loop is a part of code of a program which is executed repeatedly.
- A loop is used using condition. The repetition is done until condition becomes condition true.
- A loop declaration and execution can be done in following ways.
  - Check condition to start a loop.
  - Initialize loop with declaring a variable.
  - Executing statements inside loop.
  - Increment or decrement of value of a variable.
- **Types of looping statements:** Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as: Before loop and After loop. So, there are two types of looping statements.

1. **Entry controlled loop:** In such type of loop, the test condition is checked first before the loop is executed. Some common examples of this looping statements are while loop and for loop.

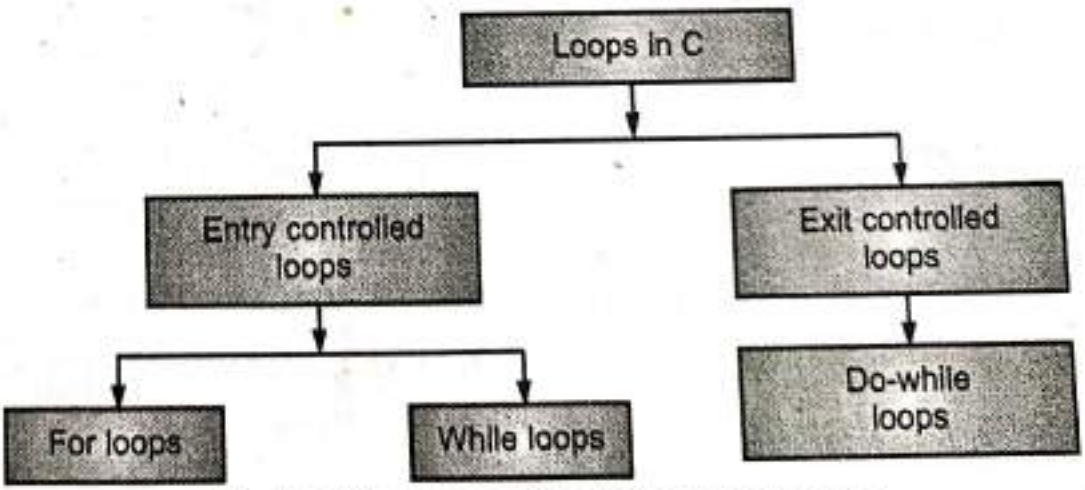


Fig. 3.9 (a): Types of Loops in C Language

2. **Exit controlled loop:** In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop must be executed atleast one time. Common example of this looping statement is a do-while loop.

3.3.1 for Loop

[W-14, 16, S-17]

- This is an entry controlled looping statement.
- In for loop structure, more than one variable can be initilized.
- One of the most important feature of this loop is that the three actions can be taken a a time such as variable initilization, condition checking and increment/decrement.
- The for loop can be more compact and flexible than that of while and do-while loops.

Syntax:

```
for(initialization; test-condition; increment/decrement)
{
    statements;
}
```

- In above syntax, the given three expressions are seperated by ; (Semicolon).

How for loop works in C programming?

- The initial expression is initialized only once at the beginning of the for loop. Then, the test expression is checked by the program. If the test expression is false, for loop is terminated.
- But, if test expression is true then, the codes are executed and expression is updated. Again, the test expression is checked. If it is false, loop is terminated and if it is true, the same process repeats until test expression is false. Fig. 3.9 (b) shows flowchart for the working of for loop in C programming.

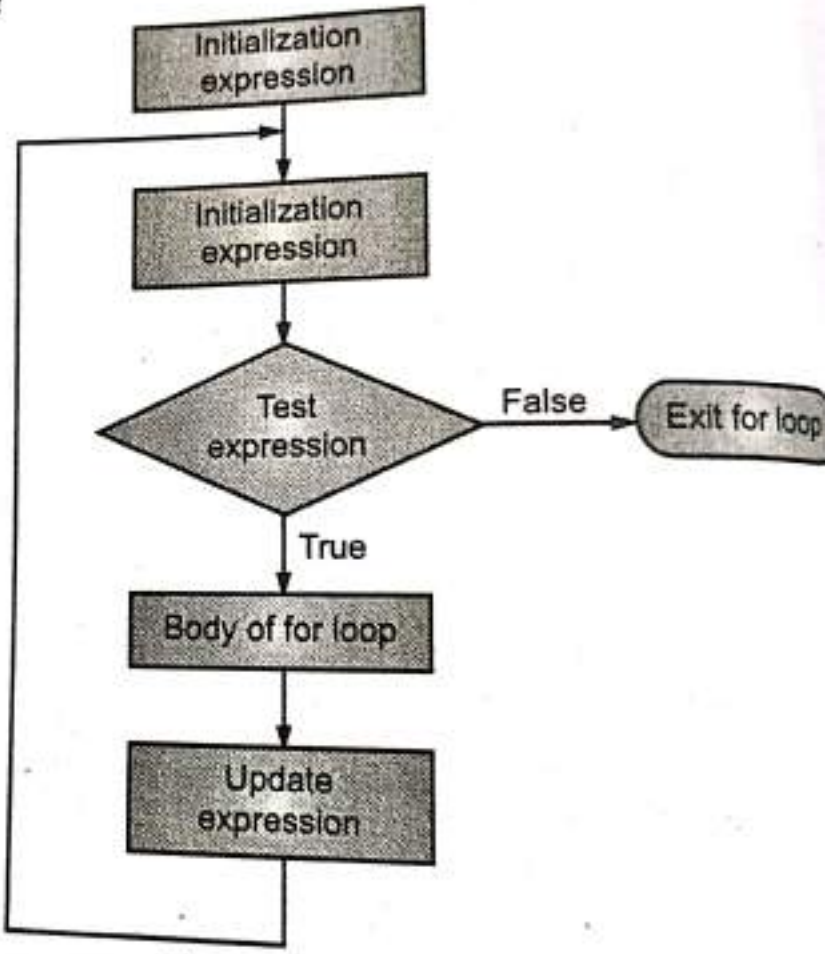


Fig. 3.9 (b): Flowchart of for loop

- **Features of for loop:**
  1. More concise,
  2. Easy to use,
  3. Highly flexible,
  4. More than one variable can be initialized,
  5. More than one increments can be applied, and
  6. More than two conditions can be used.

Program 3.13: Program for 'for' loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a i;
    clrscr();
```

```
    for(i=0; i<5; i++)
    {
        printf("\n\t Nirali Prakashan"); // 5 times
    }
    getch();
}
```

Output:

```
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
```

Program 3.14: Program to print fibonacci series. The first two Fibonacci numbers are 0 and 1 then next number is addition of previous two numbers. S-18

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.....

In mathematics it is defined by recurrence relation.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c,i,n;
    clrscr();
    a=0;
    b=1;
    printf("\n Enter number for how many times generate series");
    scanf("%d",&n);
    printf("\n FIBONACCI SERIES\n");
    printf("\t%d\t%d",a,b);
    for(i=0;i<n;i++)
    {
        c=a+b;
        a=b;
        b=c;
        printf("\t%d",c);
    }
    getch();
}
```

Output:

```
Enter number for how many times generate series 6
FIBONACCI SERIES:
0    1    1    2    3    5    8    13_
```

Program 3.15: Program to check whether a number is prime or not.

A positive integer which is only divisible by 1 and iself is known as prime number. For example, 13 is a prime number because it is only divisible by 1 and 13 but, 15 is not prime number because it is divisible by 1, 3, 5 and 15.

```
#include <stdio.h>
int main()
{
   int n, i, flag=0;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   for(i=2;i<=n/2;++i)
   {
      if(n%i==0)
      {
         flag=1;
         break;
      }
   }
   if (flag==0)
      printf("%d is a prime number.",n);
   else
      printf("%d is not a prime number.",n);
   return 0;
}
```

Output

```
Enter a positive integer: 29
29 is a prime number.
```

Program 3.16: Program to accept a number from user then print the sum of all even numbers from 1 to n and average of number from 1 to n.

```
#include<stdio.h>
#include<conio.h>
void main()
{
      int  n, i, sum=0;
      float avg;
      clrscr();
      printf("\n Enter a number:");
      scanf("%d", &n);
      for(i=1; i<=n; i++)
```

```
        if (i%2==0)
        {
          sum=sum+i;
        }
        avg=sum/n;
        printf("\nsum  of number is %d", sum);
        printf("\naverage  is %f", avg);
        getch();
    }
```

Output:

```
Enter a number 15
sum of number is 56
average is 3.00
```

Program 3.17: Program to accept a number from user that is n then print all the odd number from n to 1.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int  n,i;
    clrscr();
    printf("\n Enter any odd number : ");
    scanf("%d", &n);
    for (i=n; i>=1; i--)
    {
   printf("\t%d", i);
   i=i-1;
    }
    getch();
}
```

Output:

```
Enter a number : 21
21 19 17 15 13 11 9 7 5 3
```

3.3.2 Nested for Loop

[W-16]

- One for loop can be nested within another for loop.
- **Syntax:**

```
for (initializing; test condition; increment/decrement)
{
statement;
for (initializing; test condition; increment/decrement)
{
body of inner loop;
}
statement;
}
```

Program 3.18: Program to print "Hello !".

```
#include<stdio.h>
void main()
{
   int i, j;
   for(i=1;i<=3;i++)   /* Outer for Loop */
   {
      for(j=1;j<=2;j++)      /* Inner for Loop */
      {
         printf("Hello!\n");
      }
   }
}
```

Output:

```
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

- In above program, two for loops used. The first (outer) for loop has condition i<=3 and the second (inner) for loop has condition i<=2. The inner for loop will get executed completely for every iteration of the outer for loop.

  Hence, when i=1 j=1 , j=2 / i=2 j=1 , j=2 / i=3 j=1 , j=2 } in all there will be $3 \times 2 = 6$ iterations

  Thus, 'Hello' will get displayed 6 times.

Program 3.19: Program to print:

```
* * * *
* * * *
* * * *
* * * *
```

```
#include<stdio.h>
void main()
{
int i, j;
clrscr();
for (j=1; j<=4; j++)
{
printf("\n");
for (i=1; i<=4; i++)
{
printf("*");
}
getch();
}
```

Output:

```
* * * *
* * * *
* * * *
* * * *
```

Program 3.20: Print following right angled binary pyramid in C programming.

```
1
01
101
0101
10101
```

```
#include <stdio.h>
int main(void)
{
int i, j;
for (i = 0; i < 5; i++)
{
for (j = 0; j <= i; j++)
{
if (((i + j) % 2) == 0)
{
printf("1");
}
else {
printf("\0");
}
printf("\t");
}
printf("\n");
}
return 0;
}
```

Output:

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

Program 3.21: Program to Print FLOYD triangle as given below:

```
1
23
456
78910
```

```
#include<stdio.h>
int main()
{
int i,j,k=1;
```

```
 int range;
printf("Enter the range: ");
 scanf("%d",&range);
printf("FLOYD'S TRIANGLE: \n\n");
for(i=1;i<=range;i++)
 {
 for(j=1;j<=i;j++,k++)
 printf("%d",k);
 printf("\n");
 }
return 0;
}
```

Output:

```
Enter the range: 4
FLOYD'S TRIANGLE:
1
2 3
4 5 6
7 8 9 10
```

Program 3.22: Program to print number pyramid pattern.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i,j;
 int num;
 clrscr();
 printf("\n Enter the number of Digits:");
 scanf("%d",&num);
 for(i=0;i<=num;i++)
 {
 for(j=0;j<i;j++)
 printf("%d ",i);
 printf("\n");
 }
getch();
}
```

Output:

```
Enter the number of Digits: 4
1
2 2
3 3 3
4 4 4 4
```

Program 3.23: Print this right angled pyramid of '*' in c Using nested loops.

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,lines;
char ch = '*';
clrscr();
printf("Enter number of lines: ");
scanf("%d",&lines);
for(i=0;i <=lines;i++)
 {
 printf("\n");
 for (j=0;j < i;j++)
 printf("%c",ch);
 }
getch();
}
```

Output:

```
Enter number of lines: 6
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

3.3.3 while Loop

[W-14]

- This is an entry controlled looping statement. It is used to repeat a block of statements until condition becomes true.

Syntax:

```
while(condition)
{
    statements;
    increment/decrement;
}
```

- In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it.
- At the end of loop increment or decrement is done to change in variable value. This process continues until test condition satisfies.
- Fig. 3.10 shows flowchart for while loop.

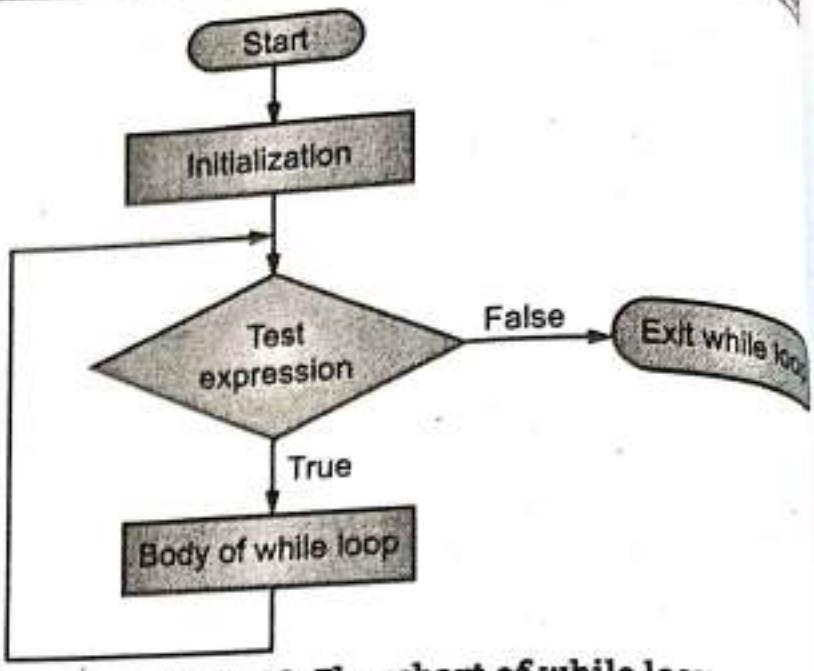


Fig. 3.10: Flowchart of while loop

Program 3.24: Program for while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a;
   clrscr();
   a=1;
   while(a<=5)
   {
      printf("\n Nirali Prakashan");
      a+=1; /* i.e. a = a + 1 */
   }
   getch();
}
```

Output:

```
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
```

Nested while loop:

- **Syntax** for nested while loop is

```
while(condition 1)
{
   statement 1;
   while (condition 2)
   statement 2;
}
```

Program 3.25: Program to print the number from 1 to 10 using a 'while' loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int num = 1;                    /* initialize counter */
while(num<=10)                  /* check the condition */
   {                            /* start of loop */
      printf("\t%d",num);       /* print the number */
      num++;                    /* increment the counter */
   }                            /* end of loop */
getch();
}
```

Output:

```
1   2   3   4   5   6   7   8   9   10
```

Program 3.26: Program to print the addition of all the numbers from 10 to 20 using a 'while' loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x = 10, sum = 0;                    /* initialize variables */
while(x <=20)                           /* check the condition */
   {                                    /* start of the loop */
      sum = sum + x;                    /* add the values */
      x++;                              /* increment counter */
}                                       /* end of the loop */
printf("\nAddition is : %d",sum);       /* print addition */
getch();
}
```

Output:

```
Addition is : 165
```

Program 3.27: Program to print the digits in reverse order.

```
/*print the digits in reverse order*/
#include <stdio.h>
#include <conio.h>
void main()
{
  int n,j;
  clrscr();
  printf("Enter the No :- ");
  scanf("%d",&n);
  printf("\n");
  printf("Reverse of the number is....");
```

```
    while(n>0)
      {
          j = n%10;
          printf("%d",j);
         n= n/10;
      }
    getch();
}
```

Output:

```
Enter the No :- 23456
Reverse of the number is....65432
```

3.3.4 do-while Loop

- This is an exit controlled looping statement.
- Sometimes, there is need to execute a block of statements first then to check condition. At that time such type of a loop is used.
- In do-while, block of statements are executed first and then condition is checked.

Syntax:

```
do
{
   statements;
   (increment/decrement);
} while(condition);
```

- In above syntax, the first the block of statements are executed. At the end of loop while statement is executed. If the resultant condition is true then program control goes to evaluate the body of a loop once again. This process continues till condition becomes true. When it becomes false, then the loop terminates.
- At first codes inside body of do is executed. Then, the test expression is checked. If it is true, code inside body of do are executed again and the process continues until test expression becomes false(zero). Notice, there is semicolon in the end of while (); in do...while loop.
- Fig. 3.11 shows flowchart of do-while loop.

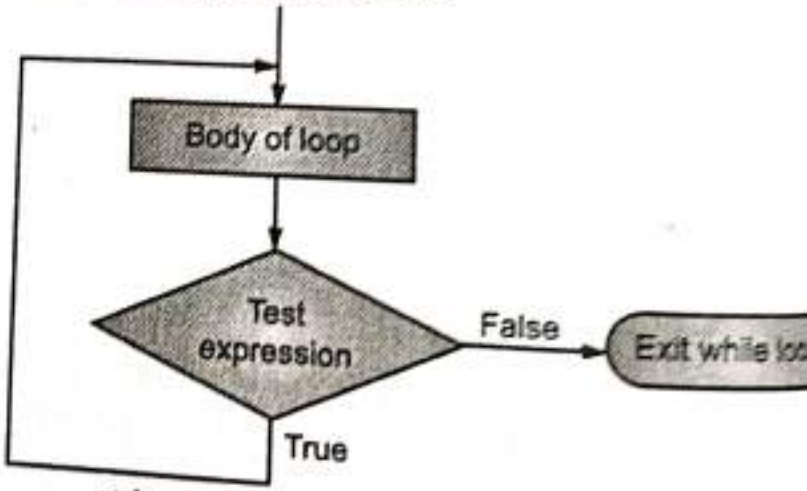


Fig. 3.11: Flowchart of do...while loop

Note: The while statement should be terminated with ; (semicolon).

Program 3.28: Program for do-while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a;
   clrscr();
   a=1;
```

```
    do
    {
        printf("\n\t Nirali Prakashan"); // 5 times
        a+=1; // i.e. a = a + 1
    }while(a<=5);
    a=6;
    do
    {
        printf("\n\n\t Pragati");     // 1 time
        a+=1; // i.e. a = a + 1
    }while(a<=5);
    getch();
}
```

Output:

```
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Nirali Prakashan
Pragati
```

Comparison between while and do-while loops:

Sr. No.	while loop	do-while loop
1.	It checks the condition at the start of the loop.	It checks the condition at the end of the loop.
2.	This is of type entry controlled loop structure.	This is of type exit controlled loop structure.
3.	The while loop is called as pre-test loop.	The do-while loop is called as post-test loop.
4.	It is not guaranteed that how many times the loop body will get executed.	The loop body will be executed at least once.

5.	**Syntax:**
	`while(condition)`
	`{`
	`  //loop body`
	`}`
		Syntax:
		`do`
		`{`
		`  //loop body`
		`}while(condition);`

3.4 JUMP STATEMENTS

- Jump statements are used to make the flow of your statements from one point to another.

3.4.1 goto Statement

- It is a well known as 'Jumping Statement.'
- The goto statement is used to change the normal sequence of program execution transferring control to other part of the program.

- The general format is,

```
goto label:
```

where label is an identifier to label the target statement to which the control will be transferred. The target must be labelled and the label must be followed by a colon.

- So the general format of target statement is:

```
label: statements;
```

- It is primarily used to transfer the control of execution to any place in a program. It is useful to provide branching within a loop.
- When the loops are deeply nested at that if an error occurs then it is difficult to get exited from such loops. Simple break statement cannot work here properly. In this situations, goto statement is used.
- Fig. 3.12 shows working of goto statement.

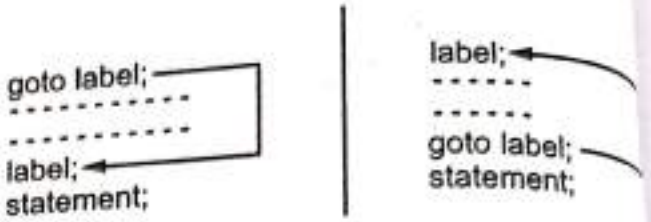


Fig. 3.12: Working of goto statement

Program 3.29: Program for goto statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i=1, j;
   clrscr();
   while(i<=3)
   {
      for(j=1; j<=3; j++)
      {
         printf(" * ");
         if(j==2)
         goto stop;
      }
      i = i + 1;
   }
   stop:
      printf("\n\n Exited !");
   getch();
}
```

Output:

```
* *

Exited!
```

Labelled Statements:

- Any statement with a label attached to it can be referred to as a labelled statement.
- Labelled statement are used as targets/way points for jump and selection statement.
- The labelled statement are represented as:

```
<label>:statements;
```

- Label points to the statement to executed next.

3.4.2 break Statement

[S-17, 18]

- Sometimes, it is necessary to exit immediately from a loop as soon as the condition is satisfied.
- When break statement is used inside a loop, then it can cause to terminate from a loop. The statements after break statement are skipped.

Syntax:

```
break;
```

- Fig. 3.13 show flowchart for break statement.

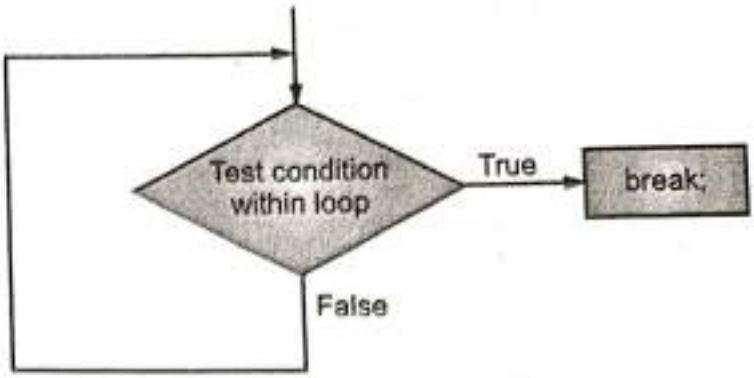


Fig. 3.13: Flowchart of break statement

Fig. 3.14 below explains the working of break statement in all three type of loops.

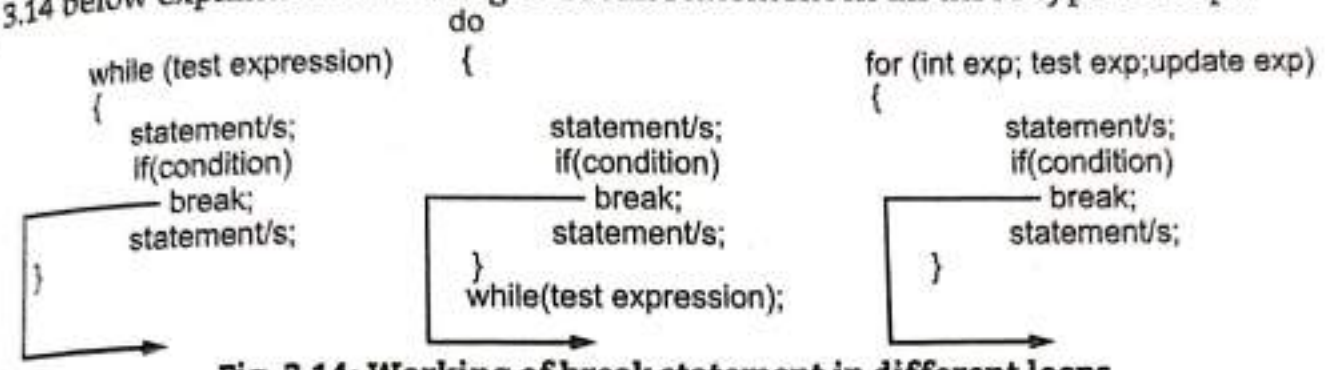


Fig. 3.14: Working of break statement in different loops

Program 3.30: Program for break statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i;
   clrscr();
   for(i=1; ; i++)
   {
      if(i>5)
      break;
      printf("%d",i); // 5 times only
   }
   getch();
}
```

Output:

```
1 2 3 4 5
```

3.4.3 continue Statement

[W-15, S-16, 17, 18]

- Sometimes, it is required to skip a part of a body of loop under specific conditions. So, C supports 'continue' statement to overcome this problem.
- The working structure of 'continue' is similar as break statement but difference is that it cannot terminate the loop.
- It causes the loop to be continued with next iteration after skipping statements in between.

- Continue statement simply skips statements and continues next iteration.
- **Syntax:** `continue;`
- Just like break, continue statement is also used with conditional if statement.
- Fig. 3.15 shows flowchart for continue statement.

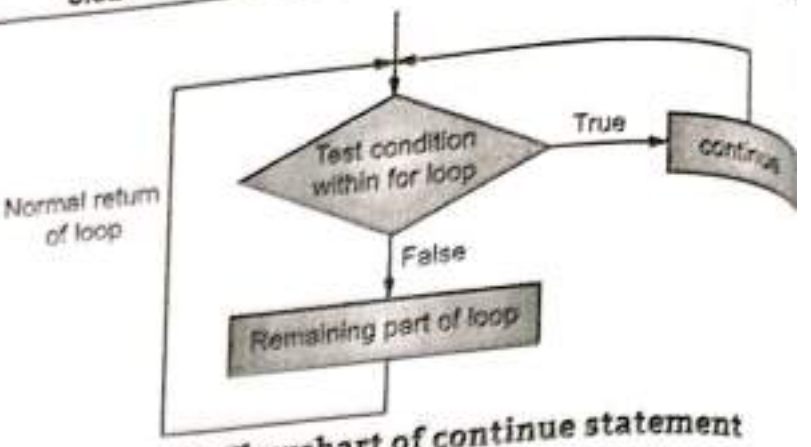


Fig. 3.15: Flowchart of continue statement

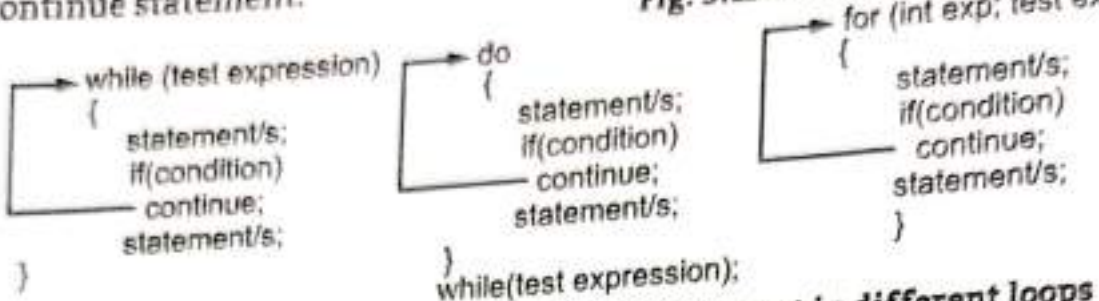


Fig. 3.16: Working of continue statement in different loops

Program 3.31: Program to demonstrate continue statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i;
   clrscr();
   for(i=1; i<=10; i++)
   {
      if(i==6)
      continue;
      printf("\t %d",i); // 6 is omitted
   }
   getch();
}
```

Output:

```
1   2   3   4   5   7   8   9   10
```

Comparison between break and continue statements:

Sr. No.	break Statement	continue Statement
1.	break statement causes loop termination.	continue statement does not causes loop termination.
2.	**Syntax:** `break;`	**Syntax:** `continue;`
3.	break statement takes control to the first statement outside the control statement.	continue statement takes control to the beginning of the loop.

contd. ...

4.	It is used in switch and loops statements.	It is only used in loop.

5.	Example:
	`int i = 4;`
	`while(i)`
	`{`
	`i--;`
	`if (i = = 2)`
	`break;`
	`printf ("%d", i);}`
	`}`
		Example:
		`int i = 4;`
		`while(i)`
		`{`
		`i--;`
		`if (i = = 2)`
		`continue;`
		`printf ("%d", i);}`
		`}`

[S-17]

3.4.4 exit() Function

- The exit() function terminates the program totally. Such forced program termination is required when an error condition is detected by the program.
- As we know the program is automatically terminated when the last statement in the main() function is executed. But, to forcefully terminate the program, the exit() function is employed.

Syntax:

```
exit (n);
```

The parameter n is an integer value, which decides the exit status.

- We can use the exit() function without 'n' also as,

```
exit ( );
```

- In such cases, the value of n is considered as 0 by default. Any non-zero of n stands for different error conditions in program.

Program 3.32: Program to print the following pattern.

```
1
12
123
1234
#include<stdio.h>
#include<conio.h>
void main()
{
   int number = 1, n, c, k;
   printf("Enter number of rows\n");
   scanf("%d",&n);
   for ( c = 1 ; c <= n ; c++ )
   {
   for( k = 1 ; k <= c ; k++ )
      {
         printf("%d ", number);
```

```
            number++;
        }
        number = 1;
        printf("\n");
    }
    getch();
}
```

Output:

```
Enter number of rows
4
1
1 2
1 2 3
1 2 3 4
```

Program 3.33: Program to print the following pattern.

```
   *
  * *
 * * *
* * * *
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, c, k = 2, j;
    printf("Enter number of rows\n");
    scanf("%d",&n);
    for ( j = 1 ; j <= n ; j++ )
    {
    for ( c = 1 ; c <= 2*n-k ; c++)
      {
          printf(" ");
      }
    k = k + 2;
    for ( c = 1 ; c <= j ; c++)
    {
    printf("*   ");
    }
        printf("\n");
    }
    getch();
}
```

Output:

```
Enter number of rows 4
       *
     *   *
   *   *   *
 *   *   *   *
```

Program 3.34: Write C program to check whether a number is armstrong or not. A number is armstrong if the sum of cubes of individual digits of a number is equal to the number itself. For example 371 is an armstrong number as $3^3 + 7^3 + 1^3 = 371$. Some other armstrong numbers are: 0, 1, 153, 370, 407. **S-16, 17**

```
#include <stdio.h>
int main()
{
 int number, sum = 0, temp, remainder;
printf("Enter an integer\n");
 scanf("%d",&number);
 temp = number;
 while( temp != 0 )
 {
 remainder = temp%10;
 sum = sum + remainder*remainder*remainder;
 temp = temp/10;
 }
if ( number == sum )
 printf("Entered number is an Armstrong number.\n");
 else
 printf("Entered number is not an Armstrong number.\n");
return 0;
}
```

Output:

```
Enter an integer
371
Entered number is an Armstrong number.
```

Summary

- 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. This is known as Branching.
- If statement used in C to check condition or to control the flow of execution of statements.
- The if-else statement, allows us to select one of the two variable options depending upon outcome of test condition.
- A nested if statement, is simply an if statement within an if statement.
- Nested if-else is a conditional statement, which is used when we want to check more than one conditions at a time in a same program.
- In C programming language, the else if ladder is a way of putting multiple ifs together when multipath decisions are involved.
- Switch statement is a multiple or multiway branching decision making statement.
- A loop is a part of code of a program which is executed repeatedly.
- for is an entry controlled looping statement. One of the most important feature of this loop is that the three actions can be taken at a time like variable initilization, condition checking and increment/decrement.

- While is an entry controlled looping statement. It is used to repeat a block of statements until condition becomes true.
- Do-while is an exit controlled looping statement. In do-while, block of statements are executed first and then condition is checked.

Check Your Understanding

1. If c is a variable initialised to 1, how many times will the following loop be executed?

```
while ((c > 0) && (c < 60))
{
loop body
c ++;
}
```

(a) 60 (b) 59
(c) 61 (d) None of these

2. A "switch" statement is used to
   (a) Switch between functions in a program.
   (b) Switch from one variable to another variable.
   (c) To choose from multiple possibilities which may arise due to different values of a single variable.
   (d) All of above.
3. Choose the correct statement.
   (a) use of goto enhances the logical clarity of the code.
   (b) use of goto makes the debugging task easier.
   (c) use goto when you want to jump out of a nested loop.
   (d) never use goto statement.
4. In a for loop, if the condition is missing then,
   (a) it is assumed to be present and taken to be false.
   (b) it is assumed to be present and taken to be true.
   (c) it results in a syntax error.
   (d) execution will be terminated abruptly.
5. Which of the following comments about for loop are correct?
   (a) Index value is retained outside the loop and can be changed from within the loop.
   (b) Body of the loop can be empty.
   (c) goto can be used to jump out of loop.
   (d) All of the above.

Answers

1. (b)	2. (c)	3. (c)	4. (b)	5. (d)

Trace The Output

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

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

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

(iv)
```
main()
{
    int c=97;
    switch(c)
    {
        case 'a':
        if(c>5)
        case 'b':
        c=100;
        printf("%d",c);
    }
}
```

Practice Questions

Q.1 Write the answer of following questions in brief:

1. What is meant by decision-making structure?
2. What is meant by loop?
3. What is the purpose of goto statement?
4. List the decision-making statements.
5. What is nested if statement.

Q.2 Write the answer of following questions:

1. Explain the for, control structures. Also, explain the role of control variables.
2. Draw a neat syntax diagram of the if statement and explain its execution with the help of suitable example.
3. Explain the purpose of switch statement. How does this structure differ from the other structures?
4. Compare switch statement with if-else statement.
5. Write a program to print following pattern:

```
*
* *
* * *
* * * *
```

6. Write a short note on the nesting of control structures.
7. Explain break and continue statement with example.

Q.3 Define terms:

1. Loop, 2. Conditional statement, 3. Compound statement, 4. Selection statement, 5. Jump Statement, 6. Null Statement, 7. Iteration Statement.

Previous Exam Questions

Winter 2014

1. What is nested structure? [2 M]

Ans. Refer to Section 3.2.3.

2. Give syntax and usage of for loop. [2 M]

Ans. Refer to Section 3.3.1.

3. Explain difference between do-while and while loop with example. [5 M]

4. Trace output [5 M]

```
main()
{
    int i = 1;
    for (; ;)
    printf("%d", i);
}
```

Ans. **Output:** Displays 1 infinite times

5. Trace output [5 M]

```
main()
{
    int i = 0, x = 0;
    for(i = 1; i < 10; i++)
    {
```

```
        if(i % 2 = = 1)
        x = x + 1;
        else
        x - -;
        printf("% d", x);
    }
};
```

Ans. **Output:** 101010101 [5 M]

6. Trace output

```
main()
{
    int j = 1;
    while()
    {
        printf("% d", j++);
        if(j > 3)
        break;
    }
}
```

Ans. **Output:** Code will display syntax error.

Winter 2015

1. What is the use of continue statement. Give example. [2 M]

Ans. Refer to Section 3.4.3.

Summer 2016

1. What is use of continue statement? Give example. [2 M]

Ans. Refer to Section 3.4.3.

2. Differentiate between entry controlled loop and exit controlled loop. [5 M]

Ans. Refer to Section 3.3.

Winter 2016

1. Give syntax of for statement. Give example. [2 M]

Ans. Refer to Section 3.3.1.

2. What is nested loop? [2 Marks]

Ans. Refer to Section 3.2.3.

3. Assume all variables have been declared and assigned values. [5 M]

```
while (count ! = 10);
{
    count = 1;
    sum = sum + X;
    count = count + 1;
}
printf ("%d", count);
printf ("%d", sum);
```

Summer 2017

1. Give example of nested for loop statement. [2 M]

Ans. Refer to Section 3.3.1.

2. Give use of break and continue statement. [2 M]

Ans. Refer to Section 3.4.

3. What is exit function? Give example. [5 M]

Ans. Refer to Section 3.3.4.

4. Explain difference between if-else and switch-case. [5 M]

Ans. Refer to Section 3.2.4.

5. Trace output [5 M]

```
main()
{
    int m = 5;
    if (m < 3) printf("%d", m+1);
    else if(m < 5) printf ("%d", m+2);
    else if(m < 7) printf ("%d", m+3);
    else printf("%d", m+4);
}
```

Ans. **Output:** 8

6. Trace output. [5 M]

```
m = 1;
do
{
    printf("%d", m);
    m = m + 2
}
while(m < 10);
```

Ans. Shows error as, m is undeclared variable.

Summer 2018

1. State any four types of statements in C. [2 M]

Ans. Refer to Section 3.1.

2. Explain the working of break and continue statements in C-language with example. [5 M]

Ans. Refer to Section 3.4.

3. Trace output [5 M]

```
main( )
{
int i = 4;
switch(i)
{
case 2 : printf("Two"); break;
case 2+2 : printf("Four ?"); break;
default : printf ("Try Again");
}
```

Ans. Four?

4...

Programs Through Conditional and Looping Statements

Learning Objectives...

- To study the various Conditional Statements.
- To apply Conditional Statements in Programs.
- To study Various Looping Statements and its Applications in Problem Solving Through Programming.
- To interpret the importance of using Conditional and Looping Statements.

4.1 INTRODUCTION [S-18]

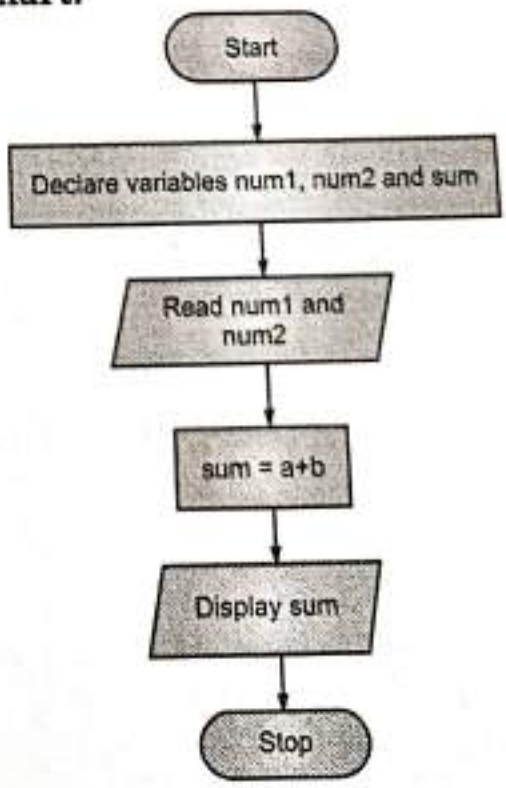
- Every procedural programming language calls for use of conditional and looping statements.
- The present chapter aims to familarise students on how to apply conditional and looping statements in programming languages.

Program 4.1: Write a program to perform addition of two integers.

Algorithm:

Step 1 : Start
Step 2 : Declare variables num1, num2 and sum.
Step 3 : Read values for num1, num2.
Step 4 : Add num1 and num2 and assign the result to a variable sum.
Step 5 : Display sum
Step 6 : Stop

Flowchart:



Program:

```
#include<stdio.h>
int main()
{
int a, b, c;
printf("Enter two numbers to
add\n");
```

```
scanf("%d%d", &a, &b);
c = a + b;
printf("Sum of the numbers = %d\n", c);
return 0;
}
```

Output:

```
Enter two numbers to add
4
5
Sum of the numbers =9
```

Program 4.2: Write a C program to perform Addition/Subtraction/Multiplication/Division operations using switch case statement.

Algorithm:

Step 1 : Start the program
Step 2 : Read the two variable num1 and num2
Step 3 : Display menu
Step 4 : Enter the option code
Step 5 : Evaluate option code with case statements
5.1: case 1 ans1=num1+num2, print ans1. goto step 7
5.2: case 2 ans1=num1-num2, print ans1. goto step 7
5.3: case 3 ans1=num1*num2, print ans1. goto step 7 Step
5.4: case 4 ans2=(float)num1/num2, print ans2. goto step 7
Step 6 : Entered case option is invalid code the print "Wrong Choice"
Step 7 : Stop the program

Program:

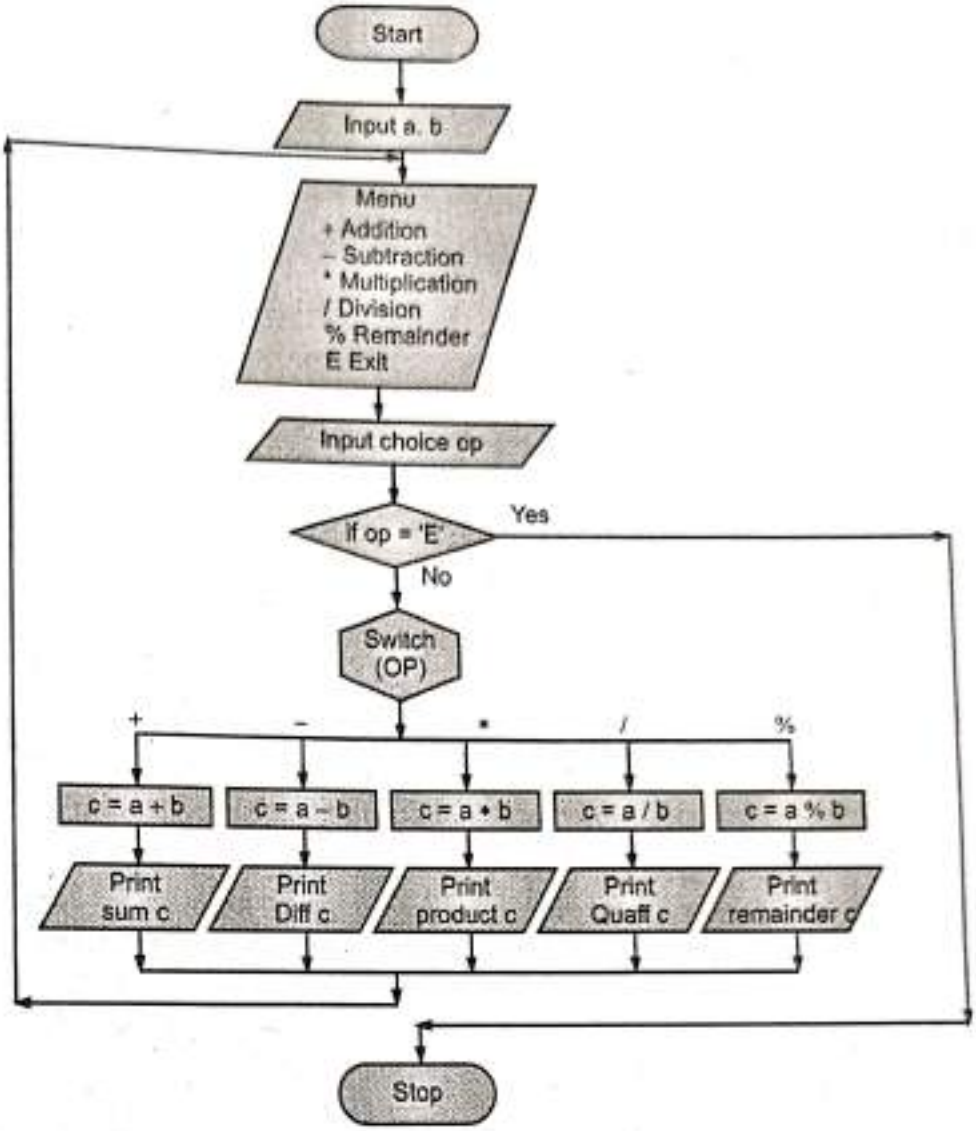
```
#include<stdio.h>
#include<conio.h>
void main()
{
int num1,num2,ans1,choice;
float ans2;
printf(" \n enter two numbers");
scanf("%d%d",&num1,&num2);
printf(" \nenter your choice \n 1.Addition \n 2.Subtraction
3.Multiplication \n 4.Division \n");
scanf("%d",&choice);
switch(choice)
{
case 1:
ans1=num1+num2;
printf("Addition =%d",ans1);
break;
```

```
case 2:
ans1=num1-num2;
printf("Subtraction
=%d",ans1);
break;
case 3:
ans1=num1*num2;
printf("Multiplicatio
n =%d",ans1);
break;
case 4:
ans2=(float)num1/num2
;
printf("Division
=%f",ans2);
break;
default:
printf("wrong
choice");
break;
}
getch();
}
```

Output:

```
enter two numbers 6 4
enter your choice
1.Addition
2.Subtraction
3.Multiplication
4.Division
1
Addition =10
```

Flowchart:

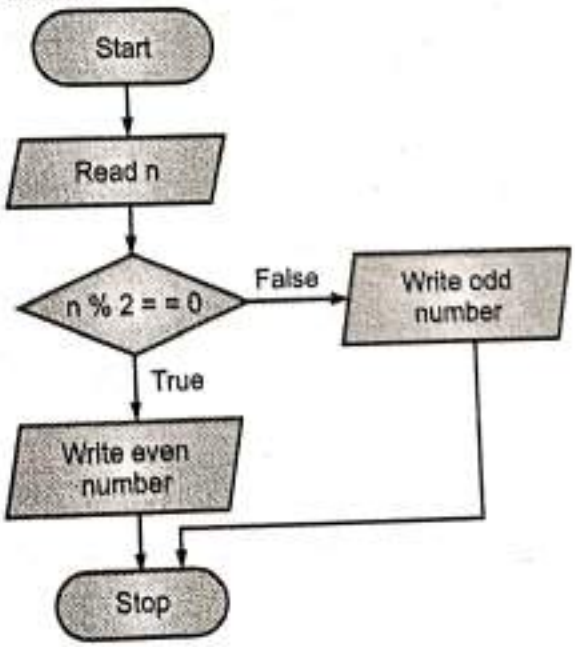


Program 4.3: Write a C program to check whether a given number is even or odd.

Algorithm:

Step 1 : Start
Step 2 : Read: Number
Step 3 : Check: If Number%2 == 0 Then
Print: N is an Even Number.
Else
Print: N is an Odd Number.
Step 4 : Exit

Flowchart:



Program:

```
#include <stdio.h>
int main()
{
int number;
printf("Enter an integer: ");
scanf("%d", &number);
```

```
// True if the number is perfectly divisible by 2
if(number % 2 == 0)
   printf("%d is even.", number);
else
   printf("%d is odd.", number);
return 0;
}
```

Output:

```
Enter an integer: -7
-7 is odd.
```

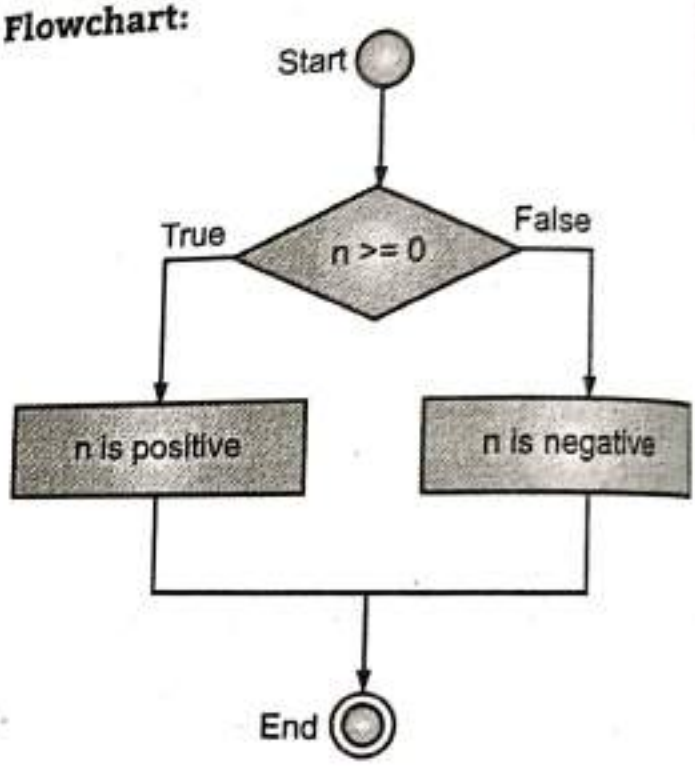
Program 4.4: Write a C program to check whether a given number is positive or negative.

In the following program, we are checking whether the input integer number is positive or negative. If the input number is greater than zero then its positive else it is negative number. If the number is zero then it is neither positive nor negative.

Algorithm:

Step 1 : START
Step 2 : Take integer variable A
Step 3 : Assign value to the variable
Step 4 : Check if A is greater than or equal to 0
Step 5 : If true print A is positive
Step 6 : If false print A is negative
Step 7 : STOP

Flowchart:



Program:

```
#include <stdio.h>
void main()
{
int num;
printf("Input a number:");
scanf("%d",&num);
if(num>=0)
printf("%d is a positive number
\n",num);
else
printf("%d is a negative number
\n",num);
}
```

Output:

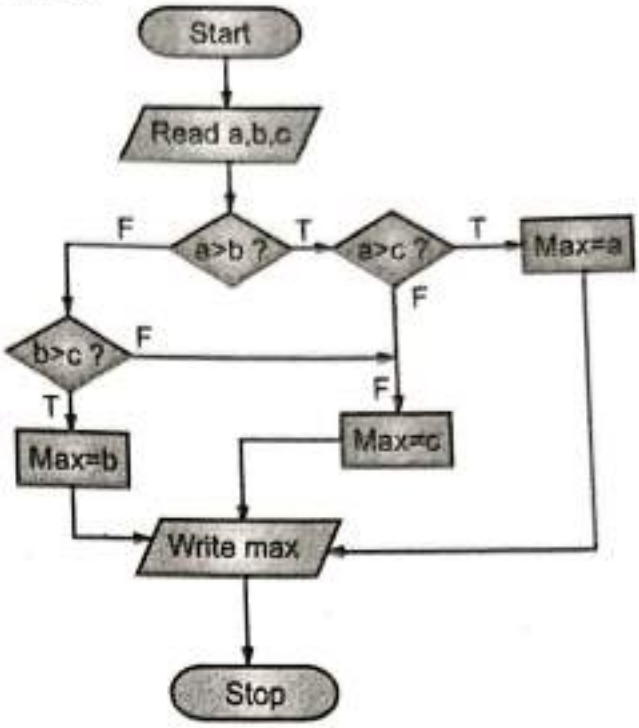
```
Input a number:15
15 is a positive number
```

Program 4.5: Write a program to find maximum of three numbers

Algorithm:

Step 1 : START
Step 2 : read a,b,c
Step 3 : if (a>b) then goto step4 else goto step 5
Step 4 : if(a>c) then max=a else max=c goto step6
Step 5 : if(b>c) then max=b else max=c goto step6
Step 6 : write max
Step 7 : STOP

Flowchart:



Program:

```
//maximum of 3 numbers
#include<stdio.h>
#include<conio.h>
void main() /*main declaration*/
{ /*varable declaration*/
int a,b,c,max;
clrscr();
printf("Enter the values of a,b,c:");
scanf("%d%d%d",&a,&b,&c);
if(a>b)
{
if(a>c)
max=a;
else
max=c;
} else if(b>c)
max=b;
else max=c;
printf("The max value is %d",max);
getch();
}
```

Output:

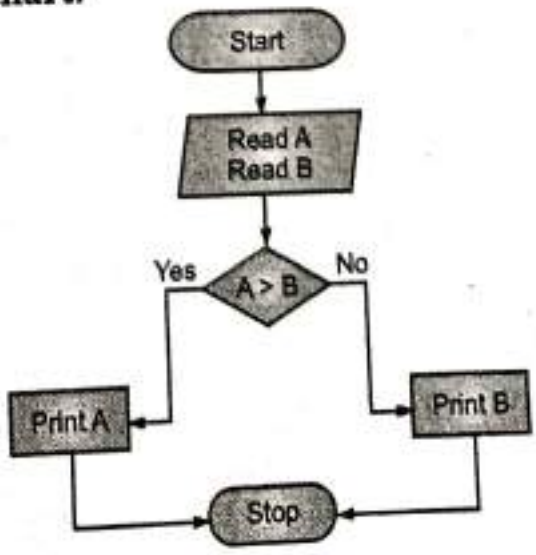
```
Enter the values of a,b,c
2 4 1
The max value is 4
```

Program 4.6: Write a program to find maximum of two numbers.

Algorithm:

Step 1 : Start
Step 2 : Read a, b .
Step 3 : If a>b then
Display "a is the largest number".
Otherwise
Display "b is the largest number".
Step 4 : Stop.

Flowchart:



Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
clrscr();
printf("Enter the values of a,b");
scanf("%d%d",&a,&b);
```

```
if(a>b)
{
printf("%d is larger number",a);
}
else
{
printf("%d is larger number",b);
}
getch();
}
```

Output:

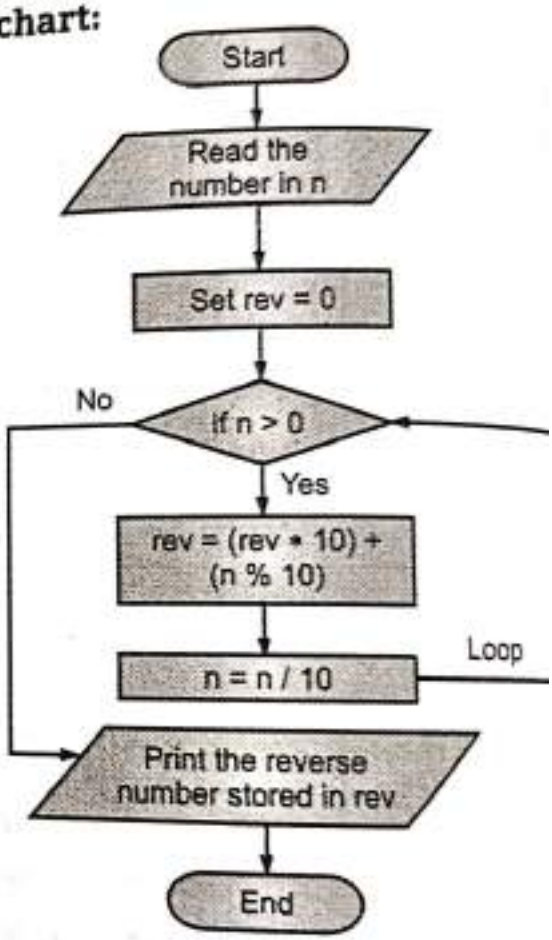
```
Enter the values of a,b
2 4
4 is larger number
```

Program 4.7: Write a program to reverse digits of a number.

Algorithm:

Step 1 : num
Step 2 : Initialize rev_num=0
Step 3 : Loop while num>0
  (a) Multiply rev_num by 10 and add remainder of num divide by 10 to rev_num rev_num=rev_num*10 + num % 10;
  (b) Divide num by 10
Step 4 : Return rev_num

Flowchart:



Program:

```
#include <stdio.h>
/* Iterative function to reverse
digits of num*/
int reversDigits(int num)
{
int rev_num = 0;
while(num> 0)
{
rev_num = rev_num*10 + num%10;
num = num/10;
}
return rev_num;
}
/*Driver program to test reversDigits*/
int main()
{
int num = 4562;
printf("Reverse of no. is %d", reversDigits(num));
getchar();
return 0;
}
```

Output:

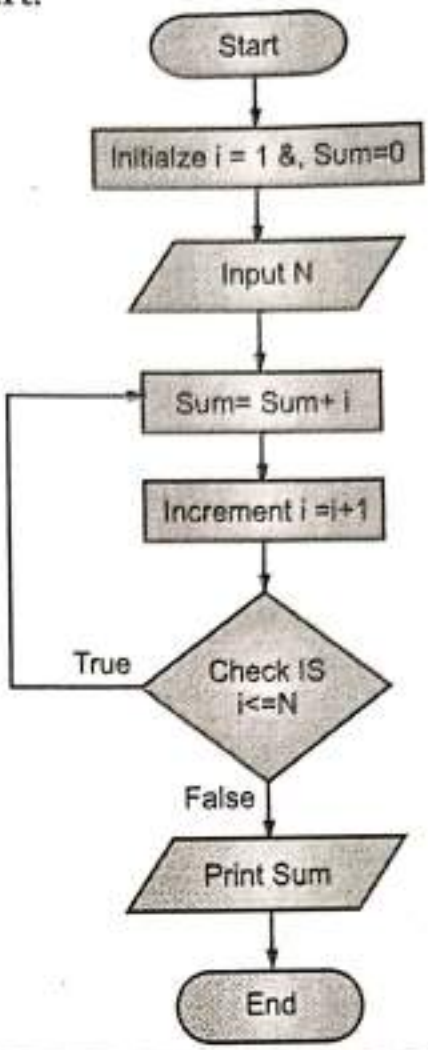
```
Reverse of no. is 2654
```

Program 4.8: Write a program on Sum of First n Numbers.

Algorithm:

Step 1 : Initialize: sum = 0
Step 2 : Run a loop from x = 1 to n and do following in loop.
sum = sum + x
Step 3 : Display sum

Flowchart:



Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int num,i,sum;
sum = 0; /* read the number */
printf("Enter number to sum:");
scanf("%d",&num);
for(i=1;i<=num;i++)
{
sum = sum + i;
}
printf("sum of %d number is
%d\n",num,sum);
getch();
return 0;
}
```

Output:

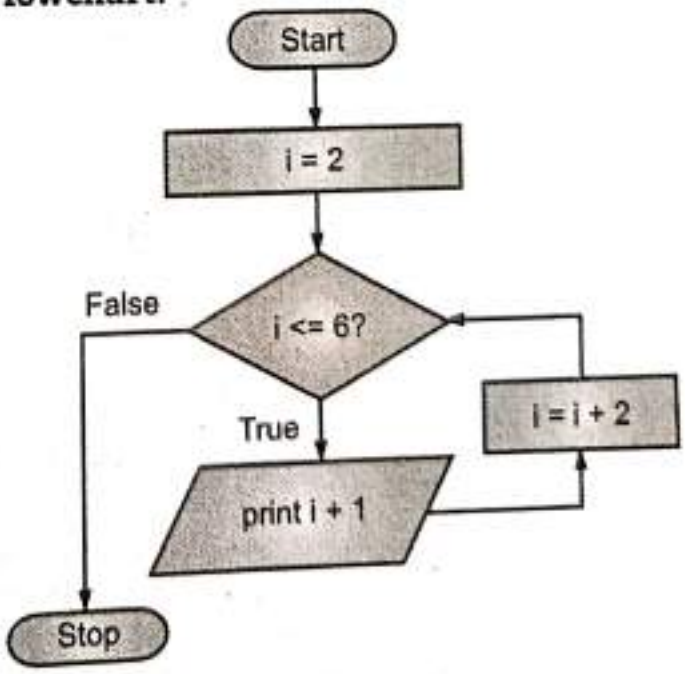
```
Enter number to sum:10
sum of 10 number is 55
```

Program 4.9: Write a program for Table Generation.

Algorithm:

Step 1 : Define table value n
Step 2 : Iterate from i = n to (n*10)
Step 3 : Display i
Step 4 : Increment i by n
Step 5 : STOP

Flowchart:



Program:

```
#include <stdio.h>
int main()
{
  int n, i;
  printf("Enter an integer: ");
  scanf("%d",&n);
  for(i=1; i<=10; ++i)
  {
  printf("%d * %d = %d \n", n, i,
  n*i);
  }
  return 0;
}
```

Output:

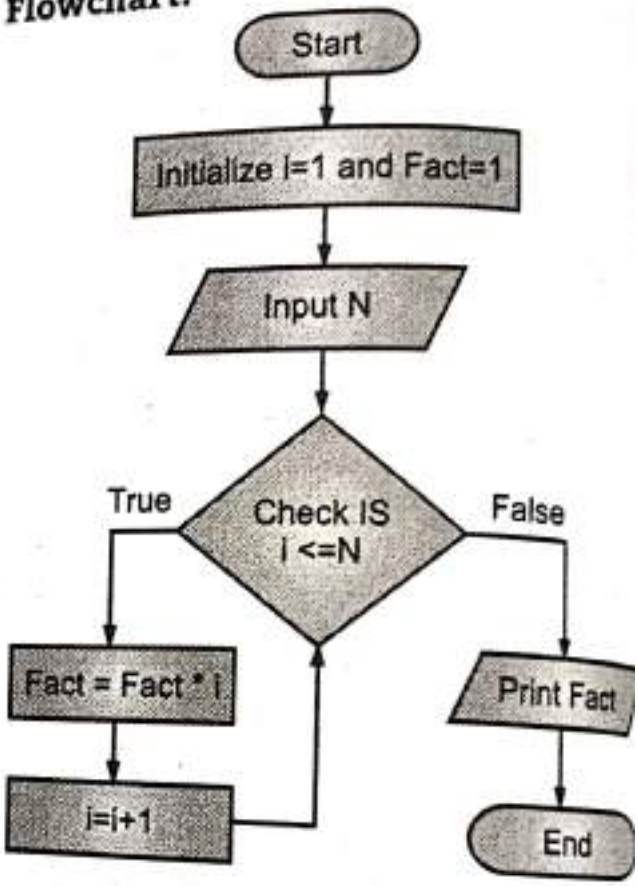
```
Enter an integer: 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
```

Program 4.10: Write a C program to find the factorial of any number entered through the keyboard.

Algorithm:

Step 1 : START
Step 2 : Initialize i=1 & Fact=1
Step 3 : Enter the any number to calculate factorial as N
Step 4 : Check is i <= N if TRUE then continue ELSE Goto Step 8
Step 5 : Fact = Fact * i
Step 6 : increment Value of i by 1. *i.e i = i+1*
Step 7 : GOTO Step 4
Step 8 : Print value of Fact
Step 9 : END

Flowchart:



Program:

```
#include<stdio.h>
void main()
{
int n,i,fact=1;
printf("Enter any number to calculate
factorial:");
scanf("%d",&n);
for(i=1;i<=n;i++)
 {
 fact=fact*i;
 }
printf("\nThe factorial of given number is %d",fact);
}
```

Output:

```
Enter any number to calculate factorial:5
The factorial of given number is 120
```

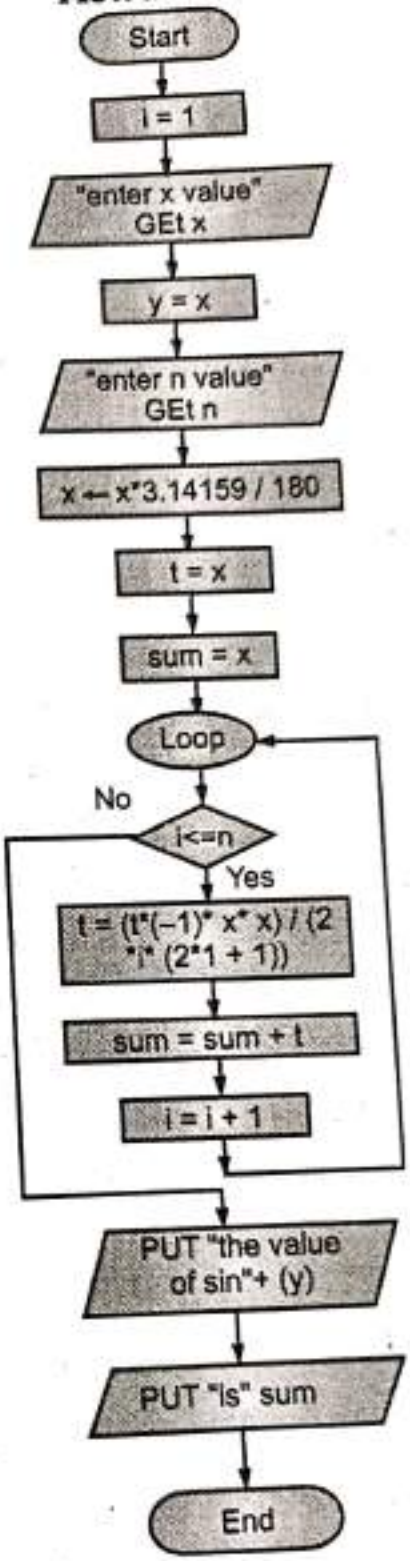
Program 4.11: Write a program for calculating the result of sine series.
sin (x) = (x^1/1!) - (x^3/3!) + (x^5/5!) - (X^7/7!) + ...

Algorithm:
Step 1 : Start
Step 2 : Initialize double sum=0,x,i,j,y,z=1, a,f=1,k=1;
Step 3 : Enter x //ie: The range of sin series
Step 4 : Read x
Step 5 : Repeat steps 6 to 11 for(i=1 to i<=x)
Step 6 : Set j=z=1
Step 7 : Repeat steps 8 while(j<=i)
Step 8 : Set z=z*i;j=j+1;
Step 9 : Repeat steps 10 while(k<=i)
Step 10: Set f=f*k;k=k+1;
Step 11: Set a=z/f;sum=sum+a;
Step 12: Display sum
Step 13: Stop

Program:

```
#include <stdio.h>
#include <math.h>
int fac(int x)
{
   int i,fac=1;
   for(i=1;i<=x;i++)
    fac=fac*i;
   return fac;
}
int main()
{
   float x,Q,sum=0;
   int i,j,limit;
   printf("Enter the value of x of sinx
   series: ");
   scanf("%f",&x);
   printf("Enter the limit upto which you
   want to expand the series: ");
   scanf("%d",&limit);
   Q=x;
   x = x*(3.1415/180);
   for(i=1,j=1;i<=limit;i++,j=j+2)
   {
    if(i%2!=0)
    {
      sum=sum+pow(x,j)/fac(j);
    }
    else
      sum=sum-pow(x,j)/fac(j);
   }
   printf("Sin(%0.1f): %f",Q,sum);
   return 0;
}
```

Flowchart:



Output:

```
Enter the value of x of sinx series: 40
Enter the limit upto which you want to expand the series: 5
Sin(40.0): 0.642772
```

Program 4.12: Write a program for calculating the result of cosine series.

Algorithm:

Step 1 : Start
Step 2 : Initialize n 20;
Step 3 : Read x
Step 4 : Convert x values into radian using formula x = x*3.1412 / 180
Step 5 : t = 1, sum = 1
Step 6 : Setup for loop from i = 1 until (i<n+1) increment i
Step 7 : t = (t*(pow(double)(1), (double) (2*i-1)) *x*x/(2*i*(2*i-1))
Step 8 : sum = sum + t
Step 9 : Print sum
Step 10: Stop

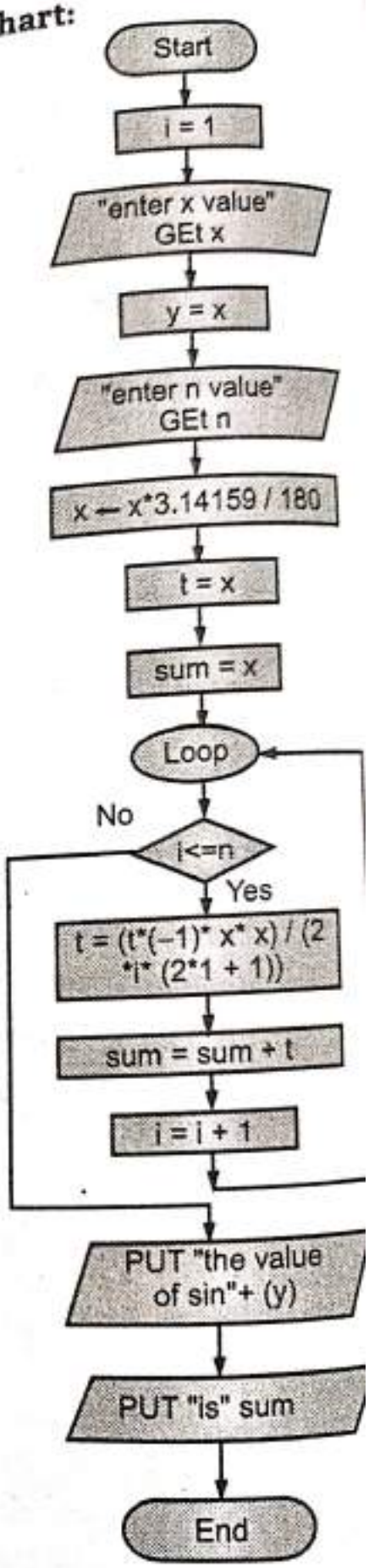
Program:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
float x, a, sum, temp;
int i, no = 20, mul;
 printf("\n enter the value of x: ");
scanf("%f", &x);
 a = x;
x = x*3.1412 / 180;
temp = 1;
sum = 1;
for(i = 1; i < no + 1; i++)
{
temp = temp *pow((double)(-1),(double)(2 *
i - 1)) * x * x / (2 * i * (2 * i -1));
sum = sum + temp;
}
printf("\n the cosine value of %f is %f",
a, sum);
 getch();
}
```

Output:

```
Enter the value of x: 60
The cosine value of 60.000000 is 0.500113
```

Flowchart:

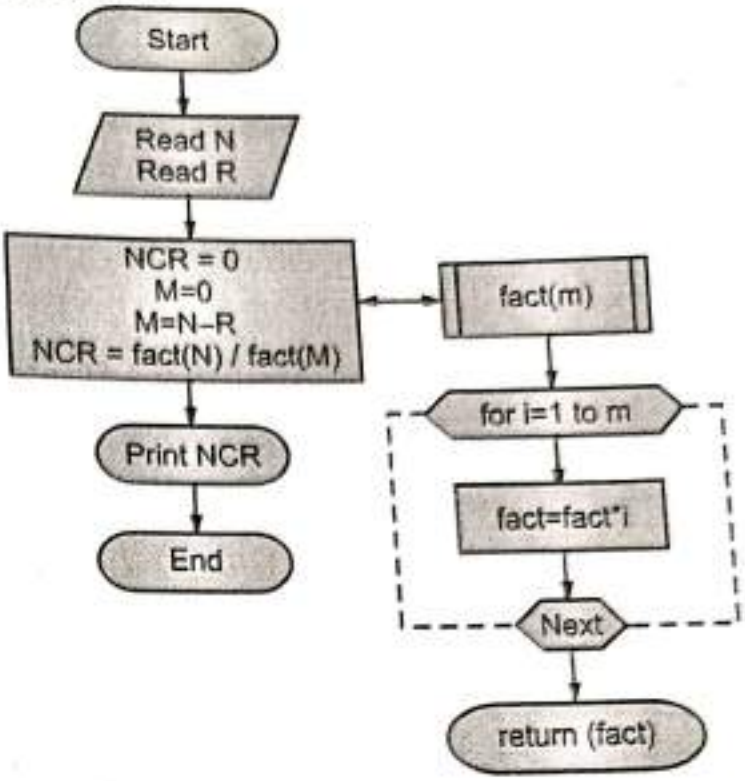


Program 4.13: Write a program to calculate nCr.

Algorithm:

step 1 : Start
step 2 : Read n, r
step 3 : take the factorial of n, r, and n – r.
step 4 : divide fact of n by multiplication of fact r and fact (n – r) and store it to result.
step 5 : return result
step 6 : End

Flowchart:



Program:

```
#include<stdio.h>
void main()
{
 int n,r,ncr;
 printf("Enter N and R values:
");
 scanf("%d %d",&n,&r);
 ncr=fact(n)/(fact(r)*fact(n-r));
printf("The NCR of %d and %d is
%d",n,r,ncr);
}
// x is a formal parameter or dummy parameter
// Calculate using n!/(n-r)!*r!
int fact(int x)
{
 int i=1;
 while(x!=0)
 {
 i=i*x;
 x--;
 }
 return i;
}
```

Output:

```
Enter N and R values: 6 3
The NCR of 6 and 3 is 20
```

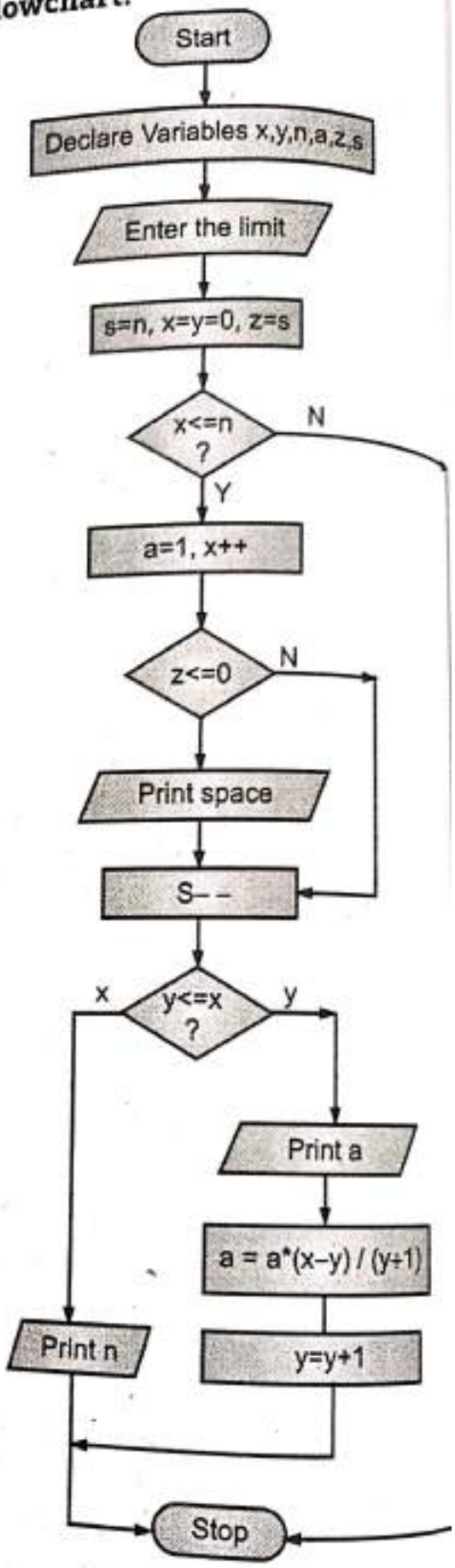
Program 4.14: Write a Program for Pascal's Triangle.

Algorithm:

Step 1 : Start
Step 2 : Declare variables x, y, n, a, z, s
Step 3 : Enter the limit
Step 4 : Initialize the value of variables, s=n , x=0, y=0 , z=s
Step 5 : Do the following operations in loop
1. x = 0 to n
2. a= 1, x++
3. z=s to 0
4. print space
5. z--
6. y = 0 to x
7. print a
8. a = a*(x-y)/(y+1)
9. y= y+1
10. go to next line

Step 6 : Repeat the process to n
Step 7 : Print the final required triangle
Step 8 : Stop

Flowchart:



Program:

```
#include <stdio.h>
long fun(int y)
{
    int z;
    long result = 1;
    for( z = 1 ; z <= y ; z++ )
    result = result*z;
    return ( result );
}
int main()
{
    int x, y, z;
    printf("Input the number of rows in
    Pascal's triangle: ");
    scanf("%d",&y);
    for ( x = 0 ; x < y ; x++ )
    {
    for ( z = 0 ; z <= ( y - x - 2 ) ;
    z++ )
    printf(" ");
    for( z = 0 ; z <= x ; z++ )
    printf("%ld  ",fun(x)/(fun(z)*fun(x-
    z)));
    printf("\n");
    }
return 0;
}
```

Output:

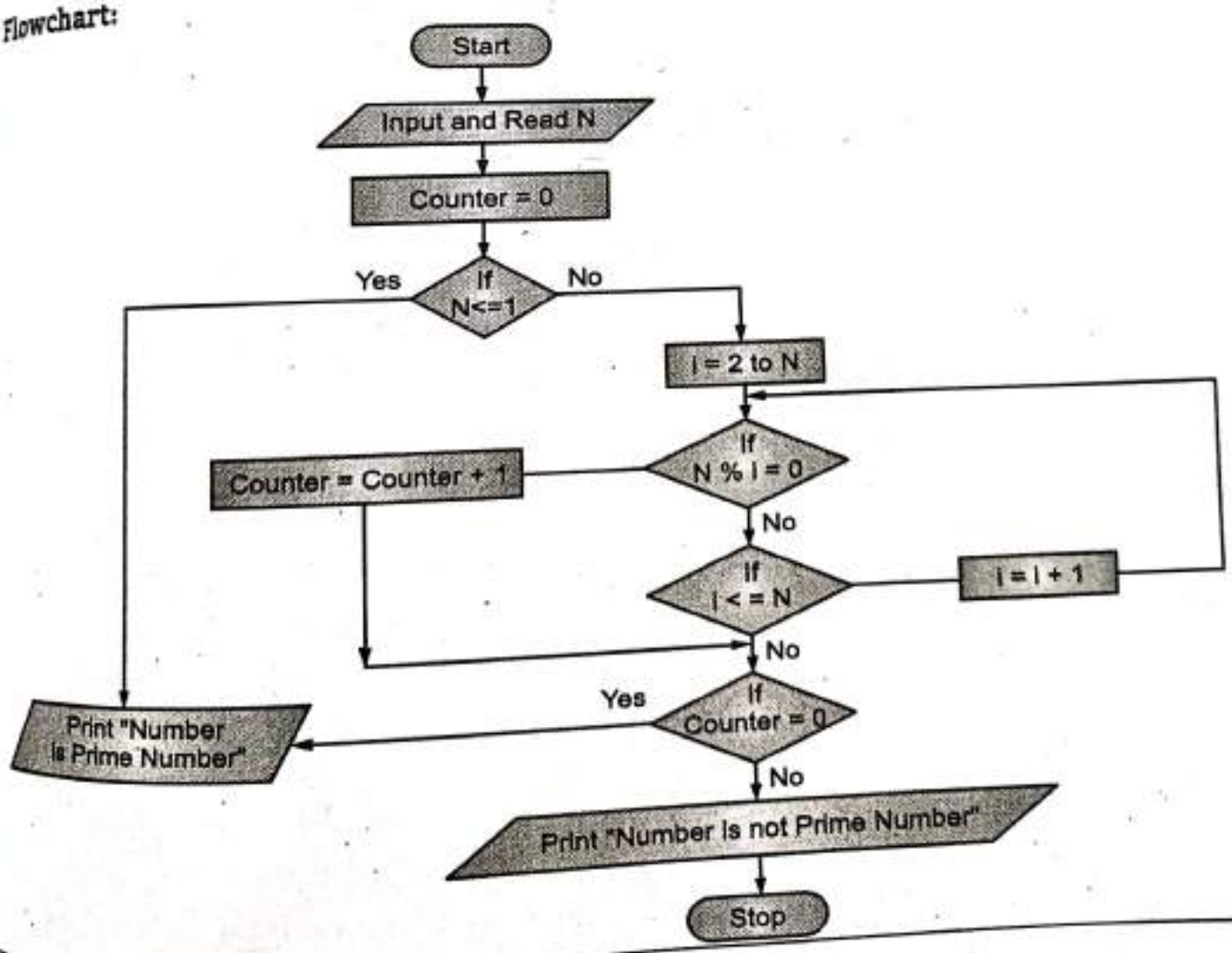
```
Input the number of rows in Pascal's triangle: 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Program 4.15: Write a program to find given integer number is prime or not and to generate prime numbers between the given range.

Algorithm:

Step 1 : Start
Step 2 : Input and Read number as N
Step 3 : Set counter = 0
Step 4 : If N < = 1
Prime 'Number is not Prime Number'
Step 5 : If for i = 2 to N and
If (N % i = 0)
counter = counter + 1, and exit
Step 6 : If counter = 0 then
Print 'Number is Prime Number'
Else
Print 'Number is not Prime Number'
Step 7 : Stop

Flowchart:



Program:

```
#include<stdio.h>
void main()
{
int gno, i=2,p=0;
printf("Enter any no to detect prime or Not:");
scanf("%d",&gno);
do
 {
 if(gno% i==0)
 p=1;
 i++;
 }while(i<gno);
if(p==0 || gno==2)
printf("\nThe Given number is prime:%d",gno);
else
printf("\nThe number is NOT prime:%d",gno);
}
```

Output:

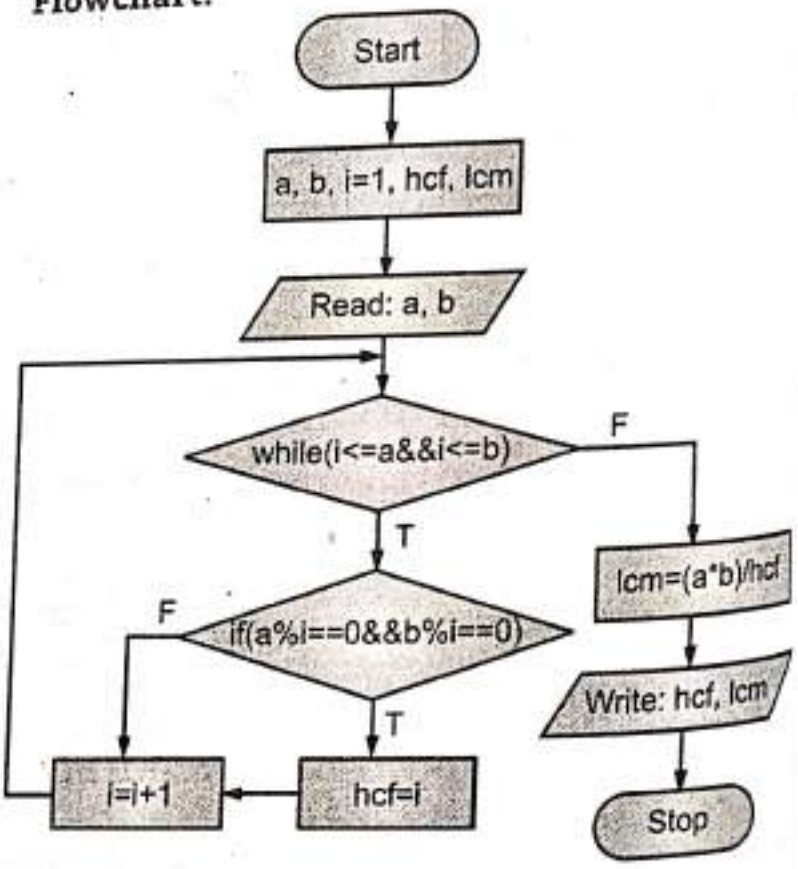
```
Enter any no to detect prime or Not:22
The number is NOT prime:22
```

Program 4.16: Write a program to find factors of a numbers

Algorithm: To find HCF and LCM:

Step 1 : Start
Step 2 : Declare variables
Step 3 : Enter the whole numbers whose LCM and HCF are to be determined
Step 4 : Equalize the variable with temporary variables
Step 5 : Divide one number by another and store the remainder in loop
Step 6 : Keep on dividing the number by successive remainder till the remainder becomes zero in loop
Step 7 : Assign the remainder with HCF
Step 8 : LCM = product of numbers / HCF
Step 9 : Print the finalized LCM and HCF
Step 10: Stop

Flowchart:



Program:

```
#include <stdio.h>
int main()
{
int m, n, a, b, t, hc, lc; // variable declaration
printf(" Enter two numbers: ");
scanf("%d%d", &a, &b);
m = a;
 n = b;
while (n != 0)
{
t = n;
n = m % n;
m = t;
}
hc = m; // hcf
lc = (a*b)/hc; // lcm
printf(" The highest Common Factor %d and %d = %d\n", a, b, hc);
printf(" The Least Common Multiple of %d and %d = %d\n", a, b, lc);
return 0;
}
```

Output:

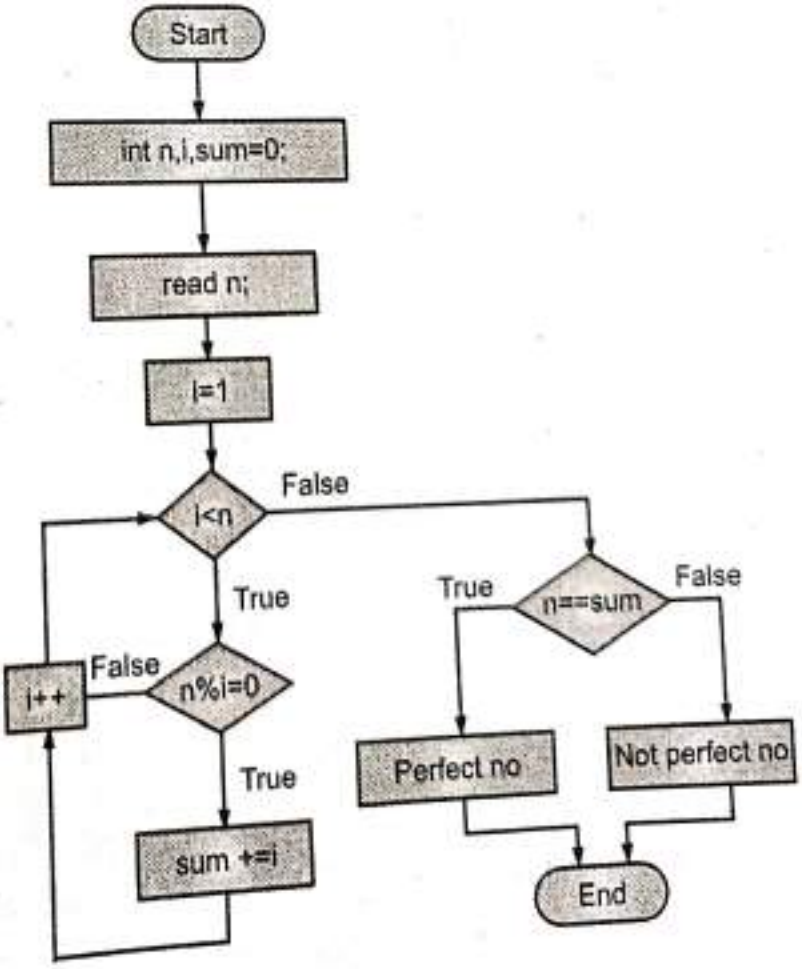
```
Enter two numbers 10 45
The highest Common Factor 10 and 45 = 5
The Least Common Multiple of 10 and 45 = 90
```

Program 4.17: Write a C Program: to Find Given Number is Perfect or Not.

Algorithm:

Step 1 : START
Step 2 : Read n
Step 3 : s = 0
Step 4 : for i=1 to n do
  (a) begin
  (b) if (n%i==0)
  (c) s=s+i
  (d) end for
Step 5 : if s==n goto step 6 else goto step 7
Step 6 : Write 'Given number is perfect number ' goto step 8
Step 7 : Write 'Given number is not a perfect number'
Step 8 : STOP

Flowchart:



Program:

```
#include<stdio.h>
#include<conio.h>
void main()
/*main declaration*/
{
 /*varible declaration*/
int i,s=0,n;
clrscr();
printf("Enter the values of n:");
scanf("%d",&n);
for(i=1;i<n;i++)
{
if(n%i==0)
{
s=s+i;
}
}
if(s==n)
printf("The given number is perfect number");
else
printf("The given number is not a perfect number");
getch();
}
```

Output:

```
Enter the values of n:28
The given number is perfect number
```

Program 4.18: Write a Program to find GCD of given numbers.

Algorithm:

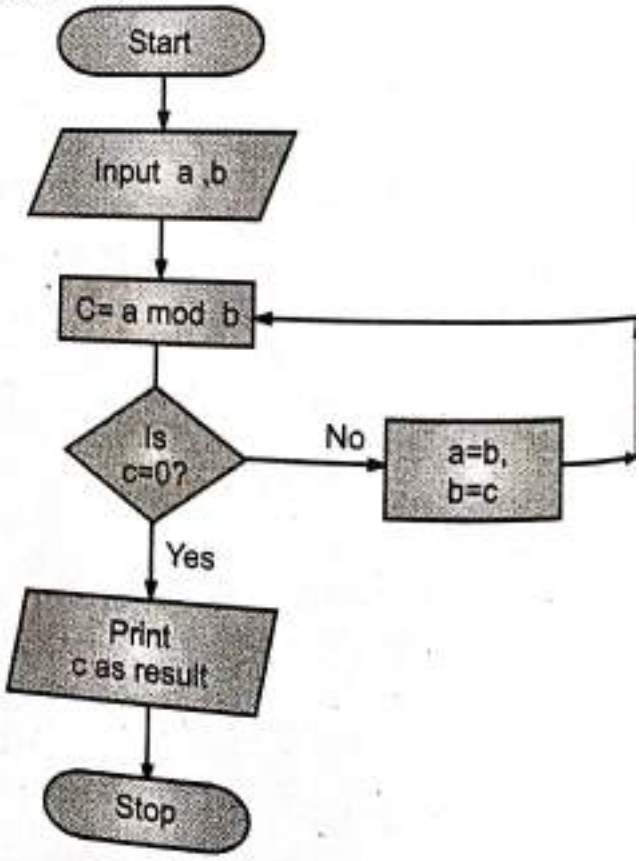
Step 1 : Read two input values using input function.

Step 2 : Convert them into integers.

Step 3 : Find smallest among two inputs, Divide both inputs by the numbers from 1 to smallest+1 If the remainders of both divisions are zero Assign that number to gcd.

Step 4 : Display the result.

Flowchart:



Program:

```
include <stdio.h>
int main()
{
   int n1, n2, i, gcd;
   printf("Enter two integers: ");
   scanf("%d %d", &n1, &n2);
```

```
    for(i=1; i <= n1 && i <= n2; ++i)
    {
      // Checks if i is factor of both integers
      if(n1%i==0 && n2%i==0)
      gcd = i;
    }
    printf("G.C.D of %d and %d is %d", n1, n2, gcd);
    return 0;
}
```

Output:

```
Enter two integers: 12 16
G.C.D of 12 and 16 is 4
```

Practice Questions

Q.1 Write a 'C' programs

1. to find sum of two numbers.
2. to find area and circumference of circle.
3. to find the simple interest.
4. to convert temperature from degree centigrade to Fahrenheit.
5. to calculate sum of 5 subjects and find percentage.
6. to show swap of two no's without using third variable.
7. to reverse a given number.
8. to find gross salary.
9. to print a table of any number.
10. to find greatest in 3 numbers.
11. to find that entered year is leap year or not.
12. to find whether given no. is even or odd.
13. to use switch statement. Display Monday to Sunday.
14. to display arithmetic operator using switch case.
15. to display first 10 natural no. & their sum.
16. to print Fibonacci series up to 100.
17. to find factorial of a number.
18. to find whether given no. is a prime no. or not.
19. to display sum of series 1+1/2+1/3+.....+1/n.
20. to display series and find sum of 1+3+5+.....+n.

Previous Exam Questions

Summer 2015

1. Write a C program to accept a four digit number from user and count zero, odd and even digits of the entered number. [5 M]

Ans. Refer to Chapter 3.

2. Write a 'C' program to accept a string from user and generate following pattern (e.g. input is string "abcd"). [5 M]

```
a
a b
a b c
a b c d
a b c
a b
a
```

Ans. Refer to Chapter 3, Program 3.48.

Winter 2015

1. Write a 'C' program to find maximum of three numbers. [5 M]

Ans. Refer to Program 4.4.

2. Write a 'C' program to accept a string from user and generate following pattern (e.g. input is string "abcd"): [5 M]

```
a
a b
a b c
a b c d
a b c
a b
a
```

Ans. Refer to Chapter 3, Program 3.48.

Summer 2016

1. Write a 'C' program to check whether a number is Armstrong or not. [5 M]

Ans. Refer to Chapter 3, Program 3.51.

2. Write a 'C' program to display the following pattern: [5 M]

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

Ans. Refer to Chapter 3, Program 3.48.

Winter 2016

1. Write a 'C' program to calculate sum of digits of a given number [e.g. 123 = 6]. [5 M]

2. Write a C program to calculate sum of the following series using function. [5 M]

$$sum = 1 + \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \frac{1}{x^4} \ldots\ldots$$

Ans. Refer to Program 4.11.

Summer 2017

1. Write a program to print all Armstrong numbers between 1 to 500. [5 M]

Ans. Refer to Chapter 3, Program 3.51.

2. Write a program to solve the following series: [5 M]

$$\frac{1^2}{1!} + \frac{2^2}{2!} + \frac{3^2}{3!} + \ldots + \frac{n^2}{n!}$$

Ans. Refer to Program 4.11.

Summer 2018

1. Write a program to generate Fibonacci series upto first 20 numbers. [5 M]

Ans. Refer to Chapter 3, Program 3.42.

2. Write a C program to generate the following output: [5 M]

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Ans. Refer to Chapter 3, Program 3.32.

5...

Arrays and Strings

Learning Objectives...

- To learn different types of Array.
- To study, how to store multiple values of same datatype in Array.
- To perform searching, sorting, insert, delete operations on array different.
- To study different operation on string.
- To learn the various operations like addition, transpose etc. of matrices.

5.1 INTRODUCTION TO ARRAYS [W-14, 16, S-16, 17, 18]

- An array is a collection of variables of the same data type and it is referenced by a common name.
- An array is a linear and homogeneous data structure.
- Linear data structure stores its individual data elements in a sequential order in the memory. Homogeneous means all individual data elements are of the same data type.
- An array is a group of data items of the same data type which share a common name.
- An array should be of a single type, comprising of integers or strings.
- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

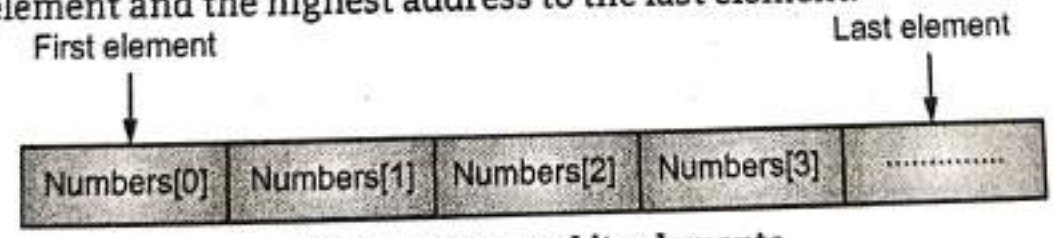


Fig. 5.1: Array and its elements

Definition of Array:

- Array is a collection of variables belongings to the same data type.

OR

- We can define array as "a variable that hold multiple elements which has the same data type".

OR

- An array is a sequence of data item of homogeneous value (same type).

- Various properties of arrays are:
  1. The size of an array is the number of elements in each dimension.
  2. The number of dimensions.
  3. The type of an array is the data type of its elements.
  4. The name of an array.

Declaration of an Array:

- Array is a collection of homogenous data stored under unique name.
- The values in an array is called as 'elements of an array.' These elements are accessed by numbers called as 'subscripts or index numbers.'
- Arrays may be of any variable type.
- Array is also called as 'subscripted variable.'
- An array is declared as,

```
data_type name_of_array [size];
```

Program 5.1: Program for displaying value entered by user.

```
#include<stdio.h>
void main()
{
   int value[5], i;
   for(i=0;i<5;i++)
   {
    printf("Enter value: ");
    scanf("%d", &value[]);
    printf("\n");
   }
   for(i=0;i<5;i++)
   {
    printf("Value = %d\n", value[i]);
    getch();
   }
}
```

Output:

```
Enter value : 0
Enter value : 1
Enter value : 2
Enter value : 3
Enter value : 4
Value = 0
Value = 1
Value = 2
Value = 3
Value = 4
```

- Here, each value entered by the user will be overwritten in the variable value. So, finally the last value '5' will be stored in value and it will get printed 5 times. So, we have to make use of an array to store 5 different integer values.

Accessing and Displaying Array Elements:

- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.
- For example:

```
double salary = balance[9];
```

- The above statement will be take 10^{th} elements from the array and assign the value of salary variable.

Program 5.2: Program for Fibonacci Series.

```
#include<stdio.h>
void main()
{
   int i, n, fibonacci[30];
   clrscr();
   printf("Enter the number of values you want: ");
   scanf("%d",&n);
   fibonacci[0] = 0;
   fibonacci[1] = 1;
   for(i=2;i<n;i++)
      fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
   printf("Fibonacci Series - \n");
    for(i=0;i<n;i++)
      printf("%d\t", fibonacci[i]);
}
```

Output:

```
Enter the number of values you want: 5
Fibonacci series: 0 1 1 2 3
```

Program 5.3: Program to find maximum number.

```
#include<stdio.h>
void main()
{
   int i, num[10], max;
   printf("Enter any 10 integer values: \n");
   for(i=0;i<10;i++)
      scanf("%d",&num[i]);
   for(i=0;i<10;i++)
   {
      if (num[i] > num[i + 1])
          num[i+1] = num[i];
   }
printf("The maximum number of the given 10 numbers is %d \t", num[9]);
}
```

Output:

```
Enter any 10 integer value:
1 3 5 8 15 14 11 12 13 2
The maximum number of the given 10 numbers is 15
```

Advantages of Arrays:

1. Easy to use.
2. Any element in an array can be accessed immediately (random access) if its exact position in the array is known.

Disadvantages of arrays are given below:

1. Elements cannot be inserted into an array.
2. Arrays do not support copy assignment.
3. Constant data type.
4. Constant size.
5. Large free sequential block to accommodate large arrays.

5.2 ONE-DIMENSIONAL ARRAY [S-16]

- The array which is used to represent and store data in a linear form is called a 'Single or One Dimensional Array.
- The arrays seen so far can be represented as a row or as a column. In other words, it can be represented as in a single dimension-width or height.

5.2.1 Definition

- An array which is having either a single row or single column is termed as a one-dimensional array. One dimensional array have the subscripts in a linear manner.

OR

- Single / One Dimensional Array is an array having a single index value to represent the arrays element.
- Fig. 5.2 shows a single dimension array or a 1D array, represented row-wise or column-wise. Such 1D arrays are also called as linear lists.

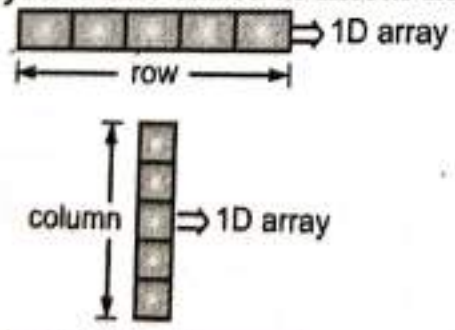


Fig. 5.2: One-dimensional array

- Now, consider a student marklist is to be maintained, specifying roll no of each student and their respective marks. In such case, we will require a 1D array for student roll no and 1D arrays for their respective subject marks. It is pretty difficult to work on such n arrays.

5.2.2 Declaration

- One-dimensional array also called as linear list, the syntax for declaration of one-dimensional array is given below

- **Syntax** of single dimesional array are:

```
<data_type> <array_name> [size];
```

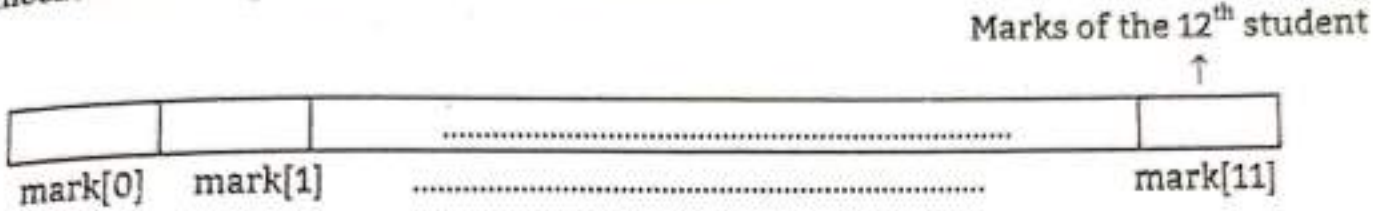
Where,

datatype : The type of the data stores in the array.
arrayname : Name of the array.
size : Maximum number of elements that an array can hold.

Example: int marks [1 2]

Brackets delimit array size
Array size
Array name
Data type

- In this example, we are representing a set of 12 student marks and the computer allocates 11 storage locations as shown below:



Example:

```
int a[3] = {2, 3, 5};
char ch[20] = "TechnoExam";
float stax[3] = {5003.23, 1940.32, 123.20};
```

Total Size (in Bytes):

```
total size = length of array * size of data type
```

- In above example, a is an array of type integer which has storage size of 3 elements. The total size would be 3 * 2 = 6 bytes.
- **Memory Allocation:**

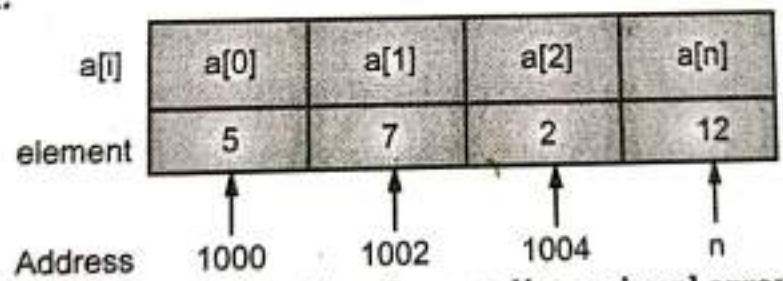


Fig. 5.3: Memory allocation for one dimensional array

Program 5.5: Program to demonstrate one dimensional array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[3], i;;
   clrscr();
   printf("\n\t Enter three numbers: ");
   for(i=0; i<3; i++)
   {
      scanf("%d", &a[i]); // read array
   }
   printf("\n\n\t Numbers are: ");
```

```
    for(i=0; i<3; i++)
    {
        printf("\t %d", a[i]); // print array
    }
    getch();
}
```

Output:

```
Enter three numbers: 9 4 6
Numbers are: 9 4 6
```

Program 5.6: Write a program to find the smallest and largest number from an array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],n,i,large,small;
printf("enter the number of elements :");
scanf("%d",&n);
printf("enter the elements :");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
large=a[0];
small=a[0];
for(i=1;i<n;i++)
{
   if(a[i]>large)
   large=a[i];
   if(a[i]<small)
   small=a[i];
}
printf("largest element in the array is : %d\n",large);
printf("smallest element in the array is : %d",small);
getch();
}
```

Output:

```
enter the number of elements : 5
enter the elements :
78  23  7  19  35
largest element in the array is : 78
smallest element in the array is : 7
```

Program 5.7: Program to reverse an array.

C program to reverse an array: This program reverses the array elements. For example is an array of integers with three elements such that

```
a[0] = 1
a[1] = 2
a[2] = 3
```

Then on reversing the array will be,

```
a[0] = 3
a[1] = 2
a[2] = 1
```

```
#include <stdio.h>
int main()
{
int n, c, d, a[100], b[100];
printf("Enter the number of elements in array\n");
scanf("%d", &n);
printf("Enter the array elements\n");
for (c = 0; c < n ; c++)
scanf("%d", &a[c]);
/*
   Copying elements into array b starting from end of array a
*/
for (c = n - 1, d = 0; c >= 0; c--, d++)
b[d] = a[c];
/*
* Copying reversed array into original.
* Here we are modifying original array, this is optional.
*/
for (c = 0; c < n; c++)
a[c] = b[c];
printf("Reverse array is\n");
for (c = 0; c < n; c++)
printf("%d\n", a[c]);
return 0;
}
```

Output:

```
Enter the number of elements in array
5
Enter the array elements
4
8
45
4568
1231
Reverse array is
1231
4568
45
8
4
```

Program 5.8: C Program to Count Even and Odd Numbers in an Array

```
#include<stdio.h>
int main()
{
    int Size, i, a[10];
    int Even[10],Even_Count=0, Odd_Count=0,odd[10];
    printf("\n Please Enter the Size of an Array(Max 10) :  ");
    scanf("%d", &Size);
    printf("\nPlease Enter the Array Elements\n");
    for(i = 0; i < Size; i++)
    {
    scanf("%d", &a[i]);
    }
    for(i = 0; i < Size; i ++)
    {
    if(a[i] % 2 == 0)
    {
      Even[Even_Count]=a[i];
      Even_Count++;
    }
    else
    {
       Odd[Odd_Count]=a[i];
      Odd_Count++;
    }
    }
    printf("Even Numbers : \n");
    for(i = 0; i < Even_Count; i ++)
    {
      printf("\n %d",Even[i]);
    }
    printf("Odd Numbers : \n");
    for(i = 0; i < Odd_Count; i ++)
    {
      printf("\n %d",Odd[i]);
    }
  return 0;
}
```

Output:

```
Please Enter the Size of an Array :
5
```

```
Please Enter the Array Elements
4
5
6
7
8
Even Numbers:
4
6
8
Odd Numbers:
5
7
```

Program 5.9: C program to find all prime numbers from the inputted array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int ar[100],i,n,j,counter;
   printf("Enter the size of the array ");
   scanf("%d",&n);
   printf("\n Now enter the elements of the array");
   for(i=0;i<n;i++)
   {
     scanf("%d",&ar[i]);
   }
   printf(" Array is -");
   for(i=0;i<n;i++)
   {
     printf("\t %d",ar[i]);
   }
   printf("\n All the prime numbers in the array are -");
   for(i=0;i<n;i++)
   {
     counter=0;
     for(j=2;j<ar[i];j++)
     {
       if(ar[i]%j==0)
       {
         counter=1;
         break;
       }
     }
```

```
        if(counter==0)
        {
          printf("\t %d",ar[i]);
        }
      }
      getch();
  }
```

Output:

```
Enter the size of the array - 5
Now enter the elements of the array - 23    98    45    101    6
Array is - 23    98    45    101    6
All the prime numbers in the array are - 23    101
```

Features:

1. Array size should be positive number only.
2. String array always terminates with null character ('\0').
3. Array elements are countered from 0 to n-1.
4. Useful for multiple reading of elements (numbers).

Disadvantages of one-dimensional array:

1. There is no easy method to initialize large number of array elements.
2. It is difficult to initialize selected elements.

5.2.3 Initialization

- User can initialize the array element one by one.

  Syntax: `array_name[index] = value;`

  Example:
  ```
  mark[0] = 40;
  mark[1] = 90;
  ..........
  ..........
  mark[11] = 91
  ```

- User can also initialize the complete array directly as:

  Syntax: `type arrayname[] = {list of values};`

  Example: `int mark[12] = {40, 90, 27, 20, 43, 56, 82, 78, 86, 80, 91};`

- This array is stored in the memory as follows:

40	90	42	57	20	43	56	82	78	86	80	91
mark[0]	mark[1]	mark[2]	mark[3]	mark[4]	mark[5]	mark[6]	mark[7]	mark[8]	mark[9]	mark[10]	mark[11]

Example: `char a[8] = {'P', 'R', 'A', 'G', 'A', 'T', 'I'};`

- This array is stored in the memory as follows:

'P'	'R'	'A'	'G'	'A'	'T'	'I'	'\0'
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

- When the compiler access a character array, it terminates it with an additional null character. When declaring character arrays, we must always allow one extra element space for the null terminator.

- The declaration and initialization of float array:
- **Example:** `float price[4] = {2.25, 0.50, 4.00, 10.4};`
- This array is stored in the memory as follows:

2.25	0.50	4.00	10.4
price[0]	price[1]	price[2]	price[3]

- Symbolic constants may also appear in array declarations:
- **Example:**
```
#define qty 12
int item[qty]; //declares items as an array of 12 elements.
```

5.3 TWO-DIMENSIONAL ARRAY [S-17]

- The array which is used to represent and store data in a tabular form is called as 'two dimensional array.' Such type of array specially used to represent data in a matrix form.

5.3.1 Definition

- Two Dimensional Array is a simple form of multi-dimensional array that stores the array elements in a row, column matrix format.

OR

- Two-dimensional array are those type of array, which has finite number of rows and finite number of columns.
- The other option is to have a single array with 2-dimension (rows and columns), which will store all the information as shown in the Fig. 5.4.

	column 0	column 1			column n
	Subject / Roll no	Maths	Computer		
row 0	4001	.....	.....	.....	.....
row 1	4002	.....	.....	.....	.....
:	:	.....	.....	.....	.....
:	:	.....	.....	.....	.....
row n	:	.....	.....	.....	.....

Fig. 5.4: Two-dimensional array

- Here, row consists of student roll no and column consists of subjects. So, the total items in this array would be,

`rows × columns`

- A two dimensional array is referred as a matrix or a table.
- A matrix has two subscripts, one denotes the row and the other denotes the column. The first dimension (i.e. the first index) might refer to the row number, and the second dimension (i.e. the second index) to the column number.
- The general declaration of a 2D array is given as,

```
type name[row] [column];
```

where, type specifies data type of the array.
name specifies name of the array.
row specifies number of rows in the array.
column specifies number of columns in the array.

- So, an array by the name 'value' of character data type having 3 rows and 1 column will be declared as,

```
char value[3][1]
```

- Now, a step further. To enter the values into a 2D array, we require nested for loops. The inner for loop indicates the column number, while the outer for loop indicates the row number.
- The rows are represented starting from 0 to n. The columns are represented starting from 0 to m. The size of the array, with n rows and m columns is given as n × m.

5.3.2 Declaration and Initialization

1. Two-dimensional Array Declaration:

Syntax: `data_type array_name[rows] [columns];`

Where,

data_type	:	The type of the data stored in the array
array_name	:	Name of the array
rows	:	Maximum number of rows in the array
columns	:	Maximum number of columns in the array

Example : `int value[3] [3];` `// implies 3 rows and 3 columns`

2. Initialization of Two Dimensional Array:

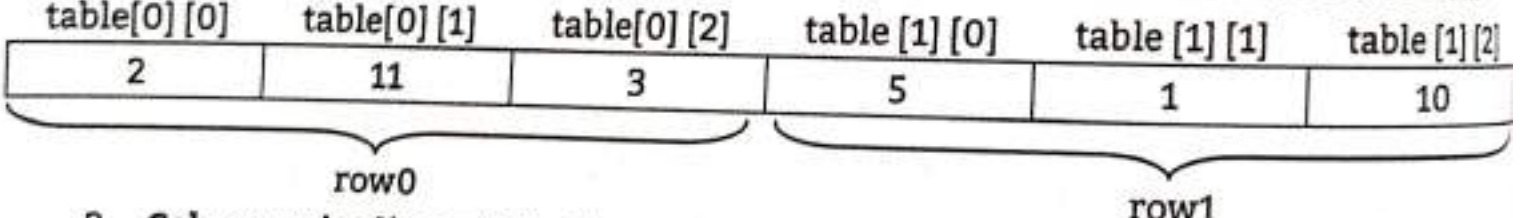
```
int table[2] [3] = {2, 11, 3, 5, 1, 10};
```

- Above initialized two-dimensional array can be stored in the memory in any of the three forms mentioned below:

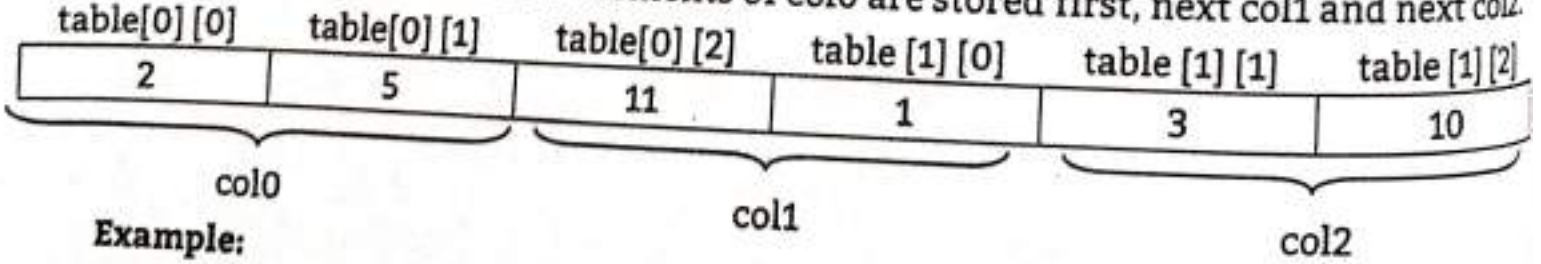
1. Two-dimensional form as follows:

	col 0	col 1	col 2
row 0	2	11	3
row 1	5	1	10

2. **Row-wise linear list:** Elements of row0 are stored first and then row1 is stored.



3. **Column-wise linear list:** Elements of col0 are stored first, next col1 and next col2.



Example:

```
int a[3][3];
```

In above example, a is an array of type integer which has storage size of 3 * 3 matrix. The total size would be 3 * 3 * 2 = 18 bytes. It is also called as 'Multidimensional Array.'

- **Memory Allocation:**

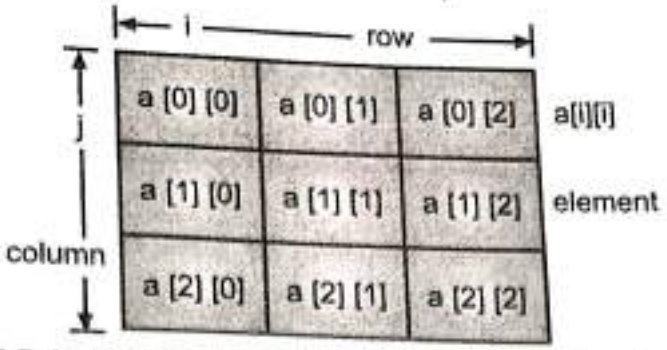


Fig. 5.5: Memory allocation for two dimensional array

Program 5.10: Program to demonstrate two dimensional array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[3][3], i, j;
   clrscr();
   printf("\n\t.Enter matrix of 3*3: ");
   for(i=0; i<3; i++)
   {
      for(j=0; j<3; j++)
      {
      scanf("%d",&a[i][j]); //read 3*3 array
      }
   }
   printf("\n\t Matrix is: \n");
   for(i=0; i<3; i++)
   {
      for(j=0; j<3; j++)
      {
      printf("\t %d",a[i][j]); //print 3*3 array
      }
    printf("\n");
   }
   getch();
}
```

Output:

```
Enter matrix of 3*3: 3 4 5 6 7 2 1 2 3
Matrix is:
3  4  5
6  7  2
1  2  3
```

- **Limitations of two dimensional array:**
  1. We cannot delete any element from an array.
  2. If we don't know that how many elements have to be stored in a memory in advance, then there will be memory wastage if large array size is specified.

Multi-dimensional array:

- Multi-dimensional arrays are very similar to standard arrays with the exception that they have multiple sets of square brackets after the array identifier.

Definition:

- An array with more than one index value is called a Multidimensional Array.

OR

- Arrays can have more than one dimension, these arrays-of-arrays are called multidimensional arrays.
- A two dimensional array can be thought of as a grid of rows and columns.
- C supports multidimensional arrays. Arrays can have higher dimensions.
- Two-dimensional arrays are identified by two subscripts, three-dimensionals arrays have three subscripts and so on.
- In general, an n-dimensional array declaration is as shown below:

```
< data type > array name [n] [m] .....
```

Program 5.11: Program for multidimensional array.

```
#include<stdio.h>
const int num_rows = 3;
const int num_columns = 5;
int main()
{
int box[num_rows][num_columns];
int row, column;
for(row = 0; row < num_rows; row++)
for(column = 0; column < num_columns; column++)
box[row][column] = column + (row * num_columns);
for(row = 0; row < num_rows; row++)
{
for(column = 0; column < num_columns; column++)
{
printf("%4d", box[row][column]);
}
printf("\n");
}
return 0;
}
```

Output:

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
```

5.4 ARRAYS AND FUNCTIONS

- Arrays can be used within different functions or they can be called and passed to and from different functions.
- Array can be passed to a function in two ways, one is element by element and another is passing the entire array.

Program 5.12: Program for display 0-5 integers using functions.

```
#include<stdio.h>
void main()
{
   void display();
   printf("First Six Integers:\n");
   display();
}
   void display()
{
   int i, num[10] = {0, 1, 2, 3, 4, 5};
   for (i=0; i<6; i++)
   {
      printf("%d",num[i]);
   }
}
```

Output:

```
First Six Integers
0 1 2 3 4 5
```

1. **Passing the array element by element:**

   In these method, the array element can be passed one-by-one to the function. The function gets access for only one element at a time and cannot modify the value.

   For example:

```
#include<stdio.h>
#include<conio.h>
main()
{
 int a[3] = {10, 20, 30};
 void display(int);
 int i;
 for (i=0; i<3; i++)
 display(a[i];
}
 void display(int b)
 {
    printf("%d", b);
 }
```

2. **Passing the entire array:**
In these method, we just have to send the name of the array to the function.
For example:

```
#include<stdio.h>
#include<conio.h>
main()
{
 int a[3] = {10, 20, 30};
 void modify(int x[3]);
 modify (a);
{
 void modify (int x[3])
 {
    int i;
    for(i=0; i<3; i++)
    x[i] = x[i] * 3;
 }
```

- One of the most important operation of array is sorting. Sorting means arranging the array element is either ascending or descending order.

Program 5.13: Program for sorting of array (ascending order).

```
#include<stdio.h>
void main()
{
   int i, j, temp;
   int ascend[10];
   printf("Enter the 10 integer values to be sorted-\n");
   for(i=0;i<10;i++)
   {
      scanf("%d",&ascend[i]);
   printf("Original Array -\n\n");
   }
   for(i=0;i<10;i++)
      {
      printf("%d\t",ascend[i]);
      }
   for(i=0;i<9;i++)
   {
      for(j=i+1;j<10;j++)
      {
          if (ascend[i] > ascend[j])
          {
               temp = ascend[i];
               ascend[i] = ascend[j];
```

```
                ascend[j] = temp;
            }
        }
    }
    printf("\nSorted Array - \n\n");
    for(i=0;i<10;i++)
    {
        printf("%d\t",ascend[i]);
    }
}
```

Output:

```
Enter the 10 integer values to be sorted -
3   6   13   -10   33  23 6   9   5   40
Sorted Array -
-10  3   5    6     6   9  13  23  33  40
```

Program 5.14: Sorting an array in descending order.

```
#include<stdio.h>
int i, j, temp;
int descend[10];
void main()
{
    void descending();
    printf("Enter the 10 integer values to be sorted - \n");
    for(i=0;i<10;i++)
    {
    scanf("%d",&descend[i]);
    }
    printf("original array");
    for(i=0;i<10;i++)
    {
    printf("%d\t",descend[i]);
    }
    descending();
    printf("\nSorted Array - \n\n");
    for(i=0;i<10;i++)
    printf("%d\t",descend[i]);
    getch();
}
void descending ()
{
    for(i=0;i<9;i++)
    {
```

```
            for (j=i+1; j<10; j++)
            {
                if (descend[i] < descend [j])
                {
                    temp = descend[i];
                    descend[i] = descend[j];
                    descend[j] = temp;
                }
            }
        }
    }
```

Output:

```
Enter the 10 integer values to be sorted:
Original Array 1  2  3  4  5  6  7  8  9  10
Sorted Array   10 9  8  7  6  5 4  3  2  1
```

Programs on Arrays

Program 1: Write a program to Input 10 numbers from user and find the total of all these numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
   int val[10], i, total=0;
   printf("Enter any ten numbers: ");
   for(i=0;i<10;i++)
   {
     scanf("%d", &val[i]);              // input numbers
     }
   for(i=0;i<10;i++)
   {
     total = total + val[i];            // find total
   }
   printf("\nTotal is: %d", total);
}
```

Output:

```
Enter any ten numbers: 10 20 30 40 50 60 70 80 90 100
Total is: 550
```

Program 2: Write a program to count the number of positive negative and zeros from given one dimensional array of n elements.

```
#include<stdio.h>
#include<conio.h>
#define size 15
```

```
void main()
{
int input[size],i,neg=0,pos=0,zro=0;n;
clrscr();
printf("how many elements you want enter(should be less than %d:",size);
scanf("%d",&n);
printf("enter elements into array:\n");
for(i=0;i<n;i++)
scanf("%d",&input[i]);
for(i=0;i<n;i++)
{
   if(input[i]>0)
   pos++;
   if(input[i]<0)
   neg++;
   if(input[i]==0)
   zro++;
}
printf("number of positive elements:%d\n",pos);
printf("number of negetive elements:%d\n",neg);
printf("number of zero elements:%d\n",zro);
getch();
}
```

Output:

```
How many elements you want enter(should be less than 15)
8
enter elements into array:
4 -8 7 13 -23 0 45 0
number of positive elements:4
number of negative elements:2
number of zero elements:2
```

Program 3: Write a program to perform addition of two 3*3 matrix.

```
/*Addition of two matrix*/
#include<stdio.h>
#include<conio.h>
void main()
{
   int x[][] = { {8,5,6},
      {1,2,1},
      {0,8,7}
            };
   int y[][] = {  {4,3,2},
      {3,6,4},
      {0,0,0}
            };
   int i,j;
   printf("First matrix:\n");
```

```
        for(i=0;i<3;i++)
        {
          for(j=0;j<3;j++)
            printf("%d", x[i][j]);
            printf("\n");
        }
        printf("Second matrix:\n");
        for(i=0;i<3;i++)
        {
          for(int j=0;j<3;j++)
            printf("%d  ", y[i][j]);
            printf("\n");
        }
        printf("Addition:\n");
        for(i=0;i<3;i++)
        {
          for(int j=0;j<3;j++)
            printf("%d  ", x[i][j]+y[i][j]);
            printf("\n");
        }
        getch();
    }
```

Output

```
First matrix:
8  5  6
1  2  1
0  8  7
Second matrix:
4  3  2
3  6  4
0  0  0
Addition:
12 8  8
4  8  5
0  8  7
```

Program 4: Program to insert an element in an array.

This code will insert an element into an array. For example consider an array a[1 having three elements in it initially and a[0] = 1, a[1] = 2 and a[2] = 3 and you want insert a number 45 at location 1 i.e. a[0] = 45, so we have to move elements one step belo so after insertion a[1] = 1 which was a[0] initially, and a[2] = 2 and a[3] = 3. Array insertio does not mean increasing its size i.e array will not be containing 11 elements.

```
#include <stdio.h>
int main()
{
 int array[100], position, c, n, value;
 printf("Enter number of elements in array\n");
 scanf("%d", &n);
 printf("Enter %d elements\n", n);
 for (c = 0; c < n; c++)
 scanf("%d", &array[c]);
 printf("Enter the location where you wish to insert an element\n");
 scanf("%d", &position);
 printf("Enter the value to insert\n");
 scanf("%d", &value);
 for (c = n - 1; c >= position - 1; c--)
 array[c+1] = array[c];
 array[position-1] = value;
 printf("Resultant array is\n");
 for (c = 0; c <= n; c++)
 printf("%d\n", array[c]);
 return 0;
}
```

Output:

```
Enter the number of elements in array
5
Enter 5 elements
2
5
4
3
8
Enter the location where you wish to insert an element
4
Enter the value to insert
10
Resultant array is
2
5
4
10
3
8
```

Program 5: Program to delete an element from an array.

```
#include <stdio.h>
int main()
{
 int array[100], position, c, n;
 printf("Enter number of elements in array\n");
 scanf("%d", &n);
 printf("Enter %d elements\n", n);
 for ( c = 0 ; c < n ; c++ )
 scanf("%d", &array[c]);
 printf("Enter the location where you wish to delete element\n");
 scanf("%d", &position);
 if ( position >= n+1 )
 printf("Deletion not possible.\n");
 else
 {
 for ( c = position - 1 ; c < n - 1 ; c++ )
 array[c] = array[c+1];
 printf("Resultant array is\n");
 for( c = 0 ; c < n - 1 ; c++ )
 printf("%d\n", array[c]);
 }
 return 0;
}
```

Output:

```
Enter number of elements in array
5
Enter 5 elements
4
6
8
10
7
Enter the location where you wish to delete element
2
Resultant array is
4
8
10
7
```

Program 6: Program to merge two arrays.

```
#include <stdio.h>
void merge(int[], int, int[], int, int[]);
int main() {
 int a[100], b[100], m, n, c, sorted[200];
 printf("Input number of elements in first array\n");
 scanf("%d", &m);
 printf("Input %d integers\n", m);
 for (c = 0; c < m; c++)
{
 scanf("%d", &a[c]);
 }
 printf("Input number of elements in second array\n");
 scanf("%d", &n);
 printf("Input %d integers\n", n);
 for (c = 0; c < n; c++)
{
 scanf("%d", &b[c]);
 }
 merge(a, m, b, n, sorted);
 printf("Sorted array:\n");
 for (c = 0; c < m + n; c++)
{
 printf("%d\n", sorted[c]);
 }
 return 0;
}
void merge(int a[], int m, int b[], int n, int sorted[]) {
 int i, j, k;
 j = k = 0;
 for (i = 0; i < m + n;)
{
 if (j < m && k < n)
{
 if (a[j] < b[k])
{
 sorted[i] = a[j];
 j++;
 }
 else {
 sorted[i] = b[k];
 k++;
 }
 i++;
 }
```

```
    else if (j == m)
    {
     for (; i < m + n;)
    {
     sorted[i] = b[k];
     k++;
     i++;
     }
     }
     else
    {
     for (; i < m + n;)
    {
     sorted[i] = a[j];
     j++;
     i++;
     }
     }
     }
    }
```

Output:

```
Enter number of elements in first array
3
Input 3 integers
1
4
6
Input number of elements in second array
3
Input 3 integers
-1
2
3
Sorted array:
-1
1
2
3
4
6
```

Program 7: Program to transpose a matrix.

Transpose of a matrix is obtained by interchanging rows and columns of a matrix. F example if a matrix is

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

then transpose of above matrix will be,

$$\begin{bmatrix} 1 & 2 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

When we transpose a matrix then the order of matrix changes, but for a square matrix order remains same.

```
#include <stdio.h>
int main()
{
 int m, n, c, d, matrix[10][10], transpose[10][10];
 printf("Enter the number of rows and columns of matrix ");
 scanf("%d%d",&m,&n);
 printf("Enter the elements of matrix \n");
 for( c = 0 ; c < m ; c++ )
 {
 for( d = 0 ; d < n ; d++ )
 {
 scanf("%d",&matrix[c][d]);
 }
 }
 for( c = 0 ; c < m ; c++ )
 {
 for( d = 0 ; d < n ; d++ )
 {
 transpose[d][c] = matrix[c][d];
 }
 }
 printf("Transpose of entered matrix:-\n");
 for( c = 0 ; c < n ; c++ )
 {
 for( d = 0 ; d < m ; d++ )
 {
 printf("%d\t",transpose[c][d]);
 }
 printf("\n");
 }
 return 0;
}
```

Output:

```
Enter the number of rows and columns of matrix
2
3
Enter the elements of matrix
1  2  3
4  5  6
Transpose of entered matrix
1  4
2  5
3  6
```

Program 8: Program for matrix multiplication.

```
#include <stdio.h>
int main()
{
 int m, n, p, q, c, d, k, sum = 0;
 int first[10][10], second[10][10], multiply[10][10];
 printf("Enter the number of rows and columns of first matrix\n");
 scanf("%d%d", &m, &n);
 printf("Enter the elements of first matrix\n");
 for ( c = 0 ; c < m ; c++ )
 for ( d = 0 ; d < n ; d++ )
 scanf("%d", &first[c][d]);
 printf("Enter the number of rows and columns of second matrix\n");
 scanf("%d%d", &p, &q);
 if ( n != p )
 printf("Matrices with entered orders can't be multiplied with each
other.\n");
 else
 {
 printf("Enter the elements of second matrix\n");
 for ( c = 0 ; c < p ; c++ )
 for ( d = 0 ; d < q ; d++ )
 scanf("%d", &second[c][d]);
 for ( c = 0 ; c < m ; c++ )
 {
 for ( d = 0 ; d < q ; d++ )
 {
 for ( k = 0 ; k < p ; k++ )
 {
 sum = sum + first[c][k]*second[k][d];
 }
 multiply[c][d] = sum;
 sum = 0;
 }
 }
 printf("Product of entered matrices:-\n");
 for ( c = 0 ; c < m ; c++ )
 {
 for ( d = 0 ; d < q ; d++ )
 printf("%d\t", multiply[c][d]);
 printf("\n");
 }
 }
getch();
 return 0;
}
```

Output:

```
Enter the number of rows and columns of first matrix
3
3
Enter the elements of first matrix
1  2  0
0  0  1
2  0  1
Enter the number of rows and columns of second matrix
3
3
Enter the elements of second matrix
1  1  2
2  1  1
1  2  1
Product of entered matrices:
5  3  4
1  2  1
3  4  5
```

Program 9: Write a program to program to search an element in the array.

```
#include <stdio.h>
#include<conio.h>
void main()
{
     int a[10], i, item;
     printf("\nEnter elements of an array:\n");
     for (i=0; i<=9; i++)
        scanf("%d", &a[i]);
     printf("\n Enter item to search: ");
     scanf("%d", &item);
     for (i=0; i<=9; i++)
     {
          if (item == a[i])
          {
               printf("\n Item found at location :%d", i+1);
               break;
          }
     }
     if (i > 9)
          printf("\n Item does not exist.");
    getch();
}
```

Output

```
Enter elements of an array:
12
34
54
76
23
89
3
77
5
90
Enter item to search : 77
Item found at location : 8
```

Program 10: C program to check if a matrix is symmetric or not: First, we find transpose of the matrix and then compare it with the original matrix. For a symmetric matrix AT = A.

```
#include<stdio.h>
int main()
{
  int m, n, c, d, matrix[10][10], transpose[10][10];
  printf("Enter the number of rows and columns of matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter elements of the matrix\n");
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &matrix[c][d]);
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      transpose[d][c] = matrix[c][d];
  if (m == n) /* check if order is same */
  {
    for (c = 0; c < m; c++)
    {
      for (d = 0; d < m; d++)
      {
        if (matrix[c][d] != transpose[c][d])
          break;
      }
      if (d != m)
        break;
    }
    if (c == m)
      printf("The matrix is symmetric.\n");
    else
      printf("The matrix is not symmetric.\n");
  }
```

```
    else
      printf("The matrix is not symmetric.\n");

    return 0;
  }
```

Output:

```
Enter the number of rows and columns of matrix
2 3
Enter elements of the matrix
4 5 6
5 6 4
The matrix is not symmetric.

Enter the number of rows and columns of matrix
2 2
Enter elements of the matrix
2 4
4 2
The matrix is symmetric.
```

Program 11: C Program to Display Upper Triangular Matrix.

```
#include <stdio.h>
void main()
{
   int i, j, r, c, array[10][10];
   printf("Enter the r and c value:");
   scanf("%d%d", &r, &c);
   for (i = 1; i <= r; i++)
   {
     for (j = 1; j <= c; j++)
     {
       printf("array[%d][%d] = ", i, j);
       scanf("%d", &array[i][j]);
     }
   }
   printf("\n matrix is \n");
   for (i = 1; i <= r; i++)
   {
     for (j = 1; j <= c; j++)
     {
       printf("%d", array[i][j]);
     }
     printf("\n");
   }
```

```
        for (i = 1; i <= r; i++)
        {
          printf("\n");
          for (j = 1; j <= c; j++)
          {
            if (i >= j)
            {
              printf("%d", array[i][j]);
            }
            else
            {
              printf("\t");
            }
          }
        }
        printf("\n\n");
        for (i = 1; i <= r; i++)
        {
          printf("\n");
          for (j = 1; j <= c; j++)
          {
             if (j >= i)
             {
               printf("%d", array[i][j]);
             }
             else
             {
               //printf("\t");
             }
             // printf("\n");
           }
        }
    }
```

Output:

```
Enter the r and c value:3 3
array[1][1] = 1 1 1
array[1][2] = array[1][3] = array[2][1] = 1 1 0
array[2][2] = array[2][3] = array[3][1] = 2 0 0
array[3][2] = array[3][3] =
matrix is
111
110
200

1
11
200

111
10
0
```

5.5 INTRODUCTION TO STRING [S-15, 18]

- A string is an array of characters stored in consecutive memory locations.
- In strings, the ending character is always the null character '\0'. The null character acts as a string terminator.
- A string is a collection of characters. Strings are always enlosed in double quotes as "string_constant".
- Strings are used in string handling operations such as,
  - Counting the length of a string.
  - Comparing two strings.
  - Copying one string to another.
  - Converting lowercase string to uppercase.
  - Converting uppercase string to lowercase.
  - Joining two strings.
  - Reversing string.

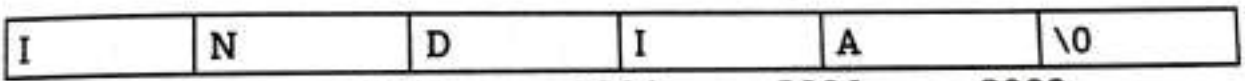
5.5.1 Definition

- String is an array of characters ended with null character ('\0').

OR

- The string is sequence of characters that is treated as a single data item. In general, the character array is called as string.
- A string is an array of characters terminated by a special character called NULL character (i.e. '\0'). Strings are always enclosed in double quotes.

Example: Suppose that the string "INDIA" is called string and it is stored in memory as:



- Each character is stored in 1 byte as its ASCII code. Since, the string is stored as an array, it is possible to manipulate individual characters using either subscript or pointer notation.
- Since "INDIA" is a string contains 5 characters. So the array word will be a 6-element array because last character of a string is "\0". The representation can be shown as:

Element No.	Subscript No.	Array Element	Corresponding data item
1	0	word [0]	I
2	1	word [1]	N
3	2	word [2]	D
4	3	word [3]	I
5	4	word [4]	A
6	5	word [5]	\0

5.5.2 Declaration and Initialization

1. **String Declaration:**

Syntax: `char string_name[length];`

Example: `char name[4];`

- This example declares as an array name which can store at the most 4 characters.

- A string is a character array, it is declared as follows:

For example:

```
char name[20];
char address[20];
```

2. String Initialization:

- Character arrays can be initialized in two ways - as individual characters or as single string.

Example:

```
char greet[10] = {'n', 'i', 'r', 'a', 'l', 'i'};
                    // This declaration automatically assigns storage
char greet[10] = "nirali"
                    // Equivalent to 7 characters including '\0' to the
                    character
char greet[ ] = "nirali";
                    // Array greet.
```

The compiler automatically stores the null character at the end of the string. The memory representation of char greet[] = "nirali" is given as follows.

n	i	r	a	l	i	\0

Program 5.12: Program to illustrate string as a character array.

```
#include<stdio.h>
void main()
{
   int i;
   char a[] = {'P','R','A','G','A','T','I','\0'};
   char b[] = "PRAGATI";
   clrscr();
   for(i=0;i<7;i++)
    printf("%c", a[i]);
   printf("\n");
   for(i=0;i<7;i++)
    printf("%c", b[i]);
   getch();
}
```

Output:

```
PRAGATI
PRAGATI
```

Program 5.13: Program to accept and display the string.

```
/*accept and display string*/
#include<stdio.h>
#include<conio.h>
void main()
{
char str[20];
```

```
clrscr();
printf("Enter name:");
scanf("%s",str);
printf("The entered name is:");
printf("%s",str);
getch();
}
```

Output:

```
Enter name: Pragati
The entered name is: Pragati
```

Program 5.14: Program to copy one string to another string.

```
/* Copy one string to other */
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10];
char str2[10];
int i=0;
clrscr();
printf("\n Enter the first string: ");
scanf("%s", str1);
printf("\n Enter the second string: ");
scanf("%s", str2);
printf("\n Strings before copying");
printf("\n First string is:%s",str1);
printf("\n Second string is:%s",str2);
for(i=0;i<10;i++)
{
   str2[i]=str1[i];
}
printf("\n Strings After copying");
printf("\n First string is:%s",str1);
printf("\n Second string is:%s",str2);
getch();
}
```

Output:

```
Enter the first string:Pragati
Enter the second string:Nirali
Strings before copying
First string is: Pragati
Second string is: Nirali
Strings After copying
First string is: Pragati
Second string is: Pragati
```

Program 5.15: Program to count number of vowels and consonants from the string.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[20];
int i=0,cnt1=0,cnt2=0;
clrscr();
printf("\n Enter the String :");
scanf("%s",str);
while(str[i]!='\0')
{
   if(str[i]=='a'  ||  str[i]=='e'  ||  str[i]=='i'  ||  str[i]=='o'
str[i]=='u' )
   {
    cnt1++;
   }
   else
   {
    cnt2++;
   }
   i++;
}
printf("\n The entered string is : %s",str);
printf("\n Vowels=%d and consonants = %d",cnt1.cnt2);
getch();
}
```

Output:

```
Enter the String : education
The entered string is : education
Vowels = 5 and consonants = 4
```

Program 5.16: Program to check whether string is palindrome or not.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str[20];
   int i=0,j=0,flag=0;
   clrscr();
   printf("\n Enter the String :");
   scanf("%s",str);
```

```
    while(str[i]!='\0')
    {
     i++;
    }
    i=i-1;
    while(j!=i)
    {
     if(str[i]==str[j])
     {
        flag=1;
     }
     else
     {
        flag=2;
        break;
     }
     j++;
     i--;
    }
    if(flag==1)
    {
     printf("%s" is palindrome",str);
    }
    else
    {
     printf("%s" is not palindrome",str);
    }
    getch();
}
```

Output:

```
Enter the String : NITIN
NITIN is palindrome
```

5.6 STRING FUNCTIONS

5.6.1 String Character related Functions

- We will see a function which deals with only single characters. These functions are include in the header file, 'ctype.h'.
- These functions are called as **character related functions**. The various functions are given in the table.

Character Related Function	Purpose
isalnum	Is the character, alphanumeric?
isalpha	Is the character, an alphabet?
isascii	Is the character, an ASCII character?
iscntrl	Is the character, a control character?
isdigit	Is the character, a digit?
isgraph	Is the character, a graphics character?
islower	Is the character, a lower case character?
isprint	Is the character, a printable character?
ispunct	Is the character, a punctuation character?
isspace	Is the character, a space?
isupper	Is the character, an upper case character?
isxdigit	Is the character, a hexadecimal digit?
tolower()	Checks whether character is alphabetic and converts to lower case.
toupper()	Checks whether character is alphabetic and converts to upper case.

- All above functions, take a single character, c as an argument. If the condition evaluates to be TRUE, it returns a non-zero value. If the condition evaluates to be FALSE, it returns a zero value.

Program 5.17: Program to illustrate isalpha() function.

```
#include<stdio.h>
#include<ctype.h>
void main()
{
    char c;
    clrscr();
    puts ("\n Enter a character");
    scanf("%c",&c);
    if (isalpha (c) > 0)
     puts ("The character is an alphabet");
    else
     puts ("The character is not an alphabet");
}
```

Output 1:

```
Enter a character
   v
The character is alphabet
```

Output 2:

```
Enter a character
   2
The character is not alphabet
```

5.6.2 Standard Library Functions

[W-14, 15, S-15, 16, 17]

- C library supports various string handling functions which are used to carry the string manipulations. These functions are declared in header file <string.h>.
- Following are some of the most commonly used functions. Consider that string1 and string2 are two string variables.

Function	Purpose
strcpy(string1, string2)	It copies/assigns the content of string2 into string1.
strncpy(string1, string2, n)	It copies only leftmost n characters from string1 into string2 .
strcat(string1, string2)	Concatenates (joins) the string2 to end of string1. String2 is appended to string1 by removing null character of string1, placing string2 from there and placing null character at the end.
strncat(string1, string2, n)	Concatenates (joins) the left most n characters of string2 to the end of string1.
strcmp(string1, string2)	Compares two strings Returns = 0 if they are equal >0 if string1 > string2 <0 if string1 < string2 The numeric difference of first non-matching characters of the strings is returned if they are not equal.
stricmp(string1, string2)	Compares two strings, ignoring the case For example: "Nirali" and "nirali" shows equal.
strncmp(string1, string2, n)	Similar to strcmp(), with difference is that, it compares only left most n characters of string1 to string2.
strlen(string1)	Finds the length of specified string i.e. number of characters in the string.
strstr(string1, string2)	It searches whether string2 is contained in string1 If found, returns the position (pointer) of the first occurrence of string2, otherwise it returns NULL pointer.
strchr(string1, ch)	It locates the first occurrence of a character ch into string string1. If found returns pointer, otherwise NULL.
strrchr(string1, ch)	It locates the last occurrence of a character ch into string string1. If found returns pointer, otherwise NULL.
strlwr(string1)	It converts the string1 to lowercase.
strupr(string1)	It converts the string1 to uppercase.
atoi(string1)	Converts string1 into an integer, returning the result. Similarly there is atol() and atof().
strset(string1,ch)	It sets character ch to all positions of string string1.
strrev ( )	Reverses the given string

- C also supports various library functions, which are dealing with single characters.

Program 5.18: Program to accept a string. Convert it to uppercase, if first character is capital letter. Otherwise convert it to lowercase.

```
#include<string.h>
#include<ctype.h>
void main()
{
    char str[20];
    clrscr();
    printf("Enter a string:");
    gets(str);
    if (isupper(str[0])) //or (str[0]>='A' && str[0]<='Z')
    strupr(str);
    else
    strlwr(str);
    printf("Converted string: %s",str);
    getch();
}
```

Output:

```
Enter a string:Nirali Prakashan
Converted string: NIRALI PRAKASHAN
Enter a string:nirali Prakashan
Converted string: nirali prakashan
```

1. strlen( ) Function:

- This string function gives the length of a string.

Syntax: `strlen(s1)`

Program 5.19: Program for strlen() function.

```
#include <stdio.h>
#include <string.h>
int main()
{
 char a[100];
 int length;
 printf("Enter a string to calculate it's length\n");
 gets(a);
 length = strlen(a);
 printf("Length of entered string is = %d\n",length);
 getch();
 return 0;
}
```

Output:

```
Enter a string to calculate it's length
programming simplified
Length of entered string is = 22
```

2. strcpy( ) Function:

- This string function used for copying one string to another string.
- **Syntax:** `strcpy (string1, string2)`

Program 5.20: Program for strcat() and strcpy() functions.

```
#include<stdio.h>
#include<string.h>
void main( )
{
    char s1[45] ;
    char blank[] = " ", c[] = "Practice !", prog[] = "Programming";
    char good[] = "Good";
    strcpy(s1,good);
    strcat(s1,blank);
    strcat(s1, prog);
    strcat(s1, blank);
    strcat(s1, c);
    puts(s1);
   getch();
}
```

Output:

```
Good Programming Practice!
```

3. strcat( ) Function:

- This string function concatenates (combine/joins) two or more strings. For joining or combining two strings strcat( ) is used.

Syntax: `strcat (string1, string2)`

Program 5.21: Program to concatenate two strings using strcat( ) function.

```
#include <stdio.h>
#include <string.h>
int main()
{
 char a[100], b[100];
 printf("Enter the first string\n");
 gets(a);
 printf("Enter the second string\n");
 gets(b);
 strcat(a,b);
 printf("String obtained on concatenation is %s\n",a);
 getch();
 return 0;
}
```

Output:

```
Enter the first string
Programming
Enter the second string
Simplified
String obtained on concatenation is Programming Simplified
```

4. strcmp( ) Function:

- This string function used for comparing two strings:

Syntax: `strcmp (string1, string2)`

Program 5.22: Program for strcmp() function.

```
#include <stdio.h>
#include <string.h>
int main()
{
 char a[100], b[100];
 printf("Enter the first string\n");
 gets(a);
 printf("Enter the second string\n");
 gets(b);
 if( strcmp(a,b) == 0 )
 printf("Entered strings are equal.\n");
 else
 printf("Entered strings are not equal.\n");
 getch();
 return 0;
}
```

Output:

```
Enter the first string
simplified the program
Enter the second string
Simplified
Entered strings are not equal.
```

Program 5.23: Program to swap two strings.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
int main()
{
 char first[100], second[100], *temp;
 printf("Enter the first string\n");
 gets(first);
 printf("Enter the second string\n");
 gets(second);
```

```
printf("\nBefore Swapping\n");
printf("First string: %s\n",first);
printf("Second string: %s\n\n",second);
temp = (char*)malloc(100);
strcpy(temp,first);
strcpy(first,second);
strcpy(second,temp);
printf("After Swapping\n");
printf("First string: %s\n",first);
printf("Second string: %s\n",second);
getch();
return 0;
}
```

Output:

```
Enter the first string
hello
Enter the second string
bye
Before Swapping
First string: hello
Second string: bye
After Swapping
First string: bye
Second string: hello
```

5.7 IMPLEMENTATION WITHOUT STANDARD LIBRARY FUNCTIONS

- Following programs describes implementation of string without using standard library functions.

Program 5.24: Program to concat two strings without using library function.

```
#include<stdio.h>
#include<string.h>
void concat(char[],char[]);
void main()
{
char s1[50],s2[30];
printf("\nEnter String 1:");
gets(s1);
printf("\nEnter String 2:");
gets(s2);
concat(s1,s2);
printf("\nConcated string is:%s",s1);
getch();
}
```

```
void concat(char s1[],char s2[])
{
int i,j;
i = strlen(s1);
for(j=0; s2[j] != ' '; i++,j++)
s1[i]=s2[j];
s1[i]=' ';
}
```

Output:

```
Enter String 1: Nirali
Enter String 2: Prakashan
Concated string is: NiraliPrakashan
```

Program 5.25: Reverse string without using library function.

```
#include<stdio.h>
#include<string.h>
void main()
{
 char str[100],temp;
 int i,j=0;
 printf("\nEnter the string:");
 gets(str);
 i=0;
 j=strlen(str)-1;
 while(i<j)
 {
 temp=str[i];
 str[i]=str[j];
 str[j]=temp;
 i++;
 j--;
 }
 printf("\nReverse string is:%s",str);
 getch();
}
```

Output:

```
Enter the string: Pragati
Reverse string is: itagarP
```

Program 5.26: Program for comparison of two strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[30],str2[30];
int i;
```

```
printf("\n Enter two strings:");
gets(str1);
gets(str2);
//loop for comparison
i=0;
while(str1[i]==str2[i] && str1[i]!=' ')
i++;
if(str1[i] > str2[i])
printf("\nstr1 > str2");
else
if(str1[i] < str2[i])
printf("\nstr1 < str2");
else
printf("\nstr1 = str2");
getch();
}
```

Output:

```
Enter two strings!
Pragati
Nirali
str1 > str2
```

Program 5.27: Program to find length of string without using library function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 char str[100];
 int length;
 printf("\nEnter the String: ");
 gets(str);
 length = 0; // Initial Length
 while(str[length]!='\0')
 length++;
 printf("\nLength of the String is: %d",length);
 getch();
}
```

Output:

```
Enter the String: Nirali
Length of String is: 6
```

Programs on String

Program 1: Program to copy contents of one array into another array.

```
/* Program to copy contents of one array into another array */
#include<stdio.h>
#include<conio.h>
void main()
{
    float a1[50], a2[50];
    int i, n;
    printf("Input the no. of elements to be used of the array \n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
     printf("Input the %d element of the array:", i+1);
     scanf("%d", &a1[i]);
    }
    for(i=0;i<n;i++)
     a2[i]=a1[i];
    printf("Print the 2nd array\n");
    for (i=0;i<n;i++)
     printf("%d\t",a2[i]);
    getch();
}
```

Output:

```
Input the no. of elements to be used of the array 3
Input the 1 element of array: 1
Input the 2 element of array: 2
Input the 3 element of array: 3
Print the 2nd array 1  2  3
```

Program 2: Program to find the number of vowels, consonants, digits and white space i a program.

```
#include<stdio.h>
int main()
{
char line[150];
int i,v,c,ch,d,s,o;
o=v=c=ch=d=s=0;
printf("Enter a line of string:\n");
gets(line);
for(i=0;line[i]!='\0';++i)
{
if(line[i]=='a' || line[i]=='e' || line[i]=='i' || line[i]=='o' ||
line[i]=='u' || line[i]=='A' || line[i]=='E' || line[i]=='I' || line[i]=='
' ||line[i]=='U')
++v;
```

```
else if((line[i]>='a'&& line[i]<='z') || (line[i]>='A'&& line[i]<='Z'))
++c;
else if(line[i]>='0'&&c<='9')
++d;
else if (line[i]==' ')
++s;
}
printf("Vowels: %d",v);
printf("\nConsonants: %d",c);
printf("\nDigits: %d",d);
printf("\nWhite spaces: %d",s);
getch();
return 0;
}
```

Output:

```
Enter a line of string:
This program is easy 2 understand
Vowels: 9
Consonants: 18
Digits: 1
White spaces: 5
```

Program 3: Convert given string into uppercase using library function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char *string = "Kamil Ajmal Khan";
printf("String before to strupr: %s\n", string);
strupr(string);
printf("String after strupr: %s\n", string);
getch();
}
```

Output:

```
String before to strupr: Kamil Ajmal Khan
String after strupr: KAMIL AJMAL KHAN
```

Program 4: Program count total number of capital and small letters from accepted.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int upper=0,lower=0;
char ch[80];
int i;
clrscr();
printf("nEnter The String: ");
gets(ch);
i=0;
```

```
while(ch[i]!='\0')
 {
 if(ch[i]>='A' && ch[i]<='Z')
 upper++;
 if(ch[i]>='a' && ch[i]<='z')
 lower++;
 i++;
 }
 printf("\nUppercase Letters: %d",upper);
 printf("\nLowercase Letters: %d",lower);
 getch();
 }
```

Output:

```
Enter The String: Kamil Khan
Uppercase Letters: 2
Lowercase Letters: 7
```

Summary

- An array is store multiple elements of similar type.
- Compiler doesn't perform bounds checking on an array. The array variable acts as a pointer to the zeroth element of the array.
- In a 1-D array, zeroth element is a single value, whereas, in a 2-D array this element is a 1-D array. On incrementing a pointer it points to the next location of its type. Array elements are stored in contiguous memory locations and so they can be accessed using pointers.
- A string is nothing but an array of characters terminated by '\0'.
- Being an array, all the characters of a string are stored in contiguous memory locations.
- scanf( ) can be used to receive multi-word strings, gets( ) can do the same job in a cleaner way.
- Both printf( ) and puts( ) can handle multi-word strings.
- Strings can be operated upon using several standard library functions like strlen( ), strcpy( ), strcat( ) and strcmp( ) which can manipulate strings. An array of pointers to strings is preferred since it takes less space and is efficient in processing strings

Check Your Understanding

1. What is right way to initialize array?
   (a) int num[6] = { 2, 4, 12, 5, 45, 5 }; (b) int n{} = { 2, 4, 12, 5, 45, 5 };
   (c) int n{6} = { 2, 4, 12 }; (d) int n(6) = { 2, 4, 12, 5, 45, 5 };
2. An array element are always stored in---------memory locations.
   (a) Sequential (b) Random
   (c) Sequential and Random (d) None of the above
3. What is the maximum number of dimensions an array in C may have?
   (a) Two (b) Eight
   (c) Sixteen
   (d) Theoretically no limit. The only practical limits are memory size and compilers

4. What will be the address of the arr[2][3] if arr is a 2-D long array of 4 rows and 5 columns and starting address of the array is 2000?
   (a) 2048 (b) 2056
   (c) 2052 (d) 2042
5. Size of the array need not be specified, when
   (a) Initialization is a part of definition
   (b) It is a formal parameter
   (c) It is a declaration
   (d) All of the above

Answers

1. (a)	2. (a)	3. (d)	4. (c)	5. (a)

Trace the Output

(a)
```
main( )
{
    int num[26], temp ;
    num[0] = 100 ;
    num[25] = 200 ;
    temp = num[25] ;
    num[25] = num[0] ;
    num[0] = temp ;
    printf ( "\n%d %d", num[0], num[25] ) ;
}
```

(b)
```
main( )
{
    int array[26], i ;
    for ( i = 0 ; i <= 25 ; i++ )
    {
        array[i] = 'A' + i ;
        printf ( "\n%d %c", array[i], array[i] ) ;
    }
}
```

(c)
```
main( )
{
    int sub[50], i ;
    for ( i = 0 ; i <= 48 ; i++ ) ;
    {
        sub[i] = i ;
        printf ( "\n%d", sub[i] ) ;
    }
}
```

(d)
```
main( )
{
    int b[ ] = { 10, 20, 30, 40, 50 } ;
    int i ;
    for ( i = 0 ; i <= 4 ; i++ )
    printf ( "\n%d" *( b + i ) ) ;
```

(e)
```
main( )
{
    static int a[5] ;
    int i ;
    for ( i = 0 ; i <= 4 ; i++ )
    printf ( "\n%d", a[i] ) ;
}
```

Practice Questions

Q.1 Write the answer of following questions in brief:

1. Is "C" a string or a character?
2. How to create an Array?
3. Can we change the size of an array at run time?
4. What is the default value of Array?
5. What is the two-dimensional array?

Q.2 Write the answer of following questions:

1. What is meant by array and string? Explain with the help of example.
2. How to declare an array? How to access an individual array element?
3. What is meant by dimensions of an array? Explain two-dimensional array with suitable example. How do you declare two-dimensional array?
4. Why nested loops are required in some applications of multidimensional arrays?
5. Write a program that uses an array to store a maximum of 20 test scores and calculate their average.

Q.3 Define terms:

1. Array, 2. String, 3. One diamensional array, 4. Two diamensional Array, 5. Multi Diamensional Array, 6. strcpy(), 7. strcmp(), 8. strcat(), 9. strlen()

Previous Exam Questions

Winter 2014

1. Explain strcat() and strcpy(). [2 M]

Ans. Refer Section 5.6.

2. Define Array. Explain how to declare and initialize two dimensional array with example. [5 M]

Ans. Refer Section 5.1.

3. Write a 'C' program for multiplication of m × n matrix. [5 M]

Ans. Refer Program 9.

4. Write a 'C' program to check if the given string is palindrome or not. [5 M]

Ans. Refer program 5.25.

5. [5 M]
```
main()
{
    Char s[] = "I am the best";
    Printf("%s", s);
    Printf("\n%c", s[3]);
    Printf("\n%c", s[8]);
}
```

Ans. **Output:**
```
I am the best
m
displays space
```

Summer 2015

1. Explain strlen() and strcpy(). [2 M]

Ans. Refer Section 5.6.

2. Define string with an example. [2 M]

Ans. Refer Section 5.5.

3. How can Array be passed to function? Explain with example. [5 M]

Ans. Refer Section 5.4.

4. Write a 'C' program to check if given string is palindrome or not using pointers. [5 M]

Ans. Refer Section 5.25.

5. [5 M]

```
Main()
{
   int a[5] = {5, 1, 15, 20, 25};
   int i,j,m;
   i=++a[1];
   j=a[1]++;
   m=a[i++];
   printf("%d %d %d",i,j,m);
   getch();
}
```

Ans. **Output:** 3, 2, 15

6. [5 M]

```
Main()
{
   char s[]="Aw what the breek";
   printf("%s",s);
   printf("ln%c",s[3]);
   printf("ln%c",s[1]);
}
```

Ans. **Output:** Aw what the breek lnwlnw

7. [5 M]

```
Main()
{
   Char s1[]="FYBCA SYBCA TYBCA";
   Char s2[20];
   Char s3[20];
   scanf(s1,"%s %s %s", s1,s2,s3);
   printf("%s, %s, %s", s1,s2,s3);
}
```

Ans. **Output:** FYBCA SYBCA TYBCA , ,

Winter 2015

1. What is two dimensional array? Explain with example. [2 M]

Ans. Refer Section 5.3. [2 M]

2. Explain strlen() and strcpy(). [5 M]

Ans. Refer Section 5.6.

3. Explain array of pointer to string with example.

Ans. Refer Section 5.7.

4. Write a 'C' program to check if given string in palindrome or not. [5 M]

Ans. Refer Program 5.25.

5. [5 M]

```
main()
{
    int a[5] = {5, 1, 15, 20, 25};
    int i, j, m;
    i = ++a[1];
    j = a[1]++;
    m=a[i++];
    printf(%d%d%d", i, j, m);
       getch();
}
```

Output: 3 2 15

Summer 2016

1. Define Array. Give example of one-dimensional array. [2 M]

Ans. Refer to Sections 5.1 and 5.2.

2. Give syntax and use of strlen() and streat(). [2 M]

Ans. Refer to Section 5.6.2.

3. [5 M]

```
#include<string.h>
#include<ctype.h>
int main(void)
{
    int length, i;
    char string[ ] = "This is A Sting";
    length = strlen(string);
    for(i=0, i<length; i + +)
    {
        string[i] = to lower(string[i]);
    }
        printf("%s\n", string);
    getch();
    return 0;
}
```

Winter 2016

1. Define string with example. [2 M]

Ans. Refer to Section 5.5.

2. What is an Array? Give an example. [2 M]

Ans. Refer to Section 5.1.

3. [5 M]

```
main()
{
    chart string[ ] = "HELLO WORLD";
    int m;
    for (m = 0; string [m]! = '\0'; m++)
       if ((m % 2) = = 0)
       printf("% c", string [m]);
}
```

4. [5 M]

```
main( )
{
   int c, d;
   char string[ ] = "A B C D";
   printf("\n\n");
   for (c = 0; c < = 11; c++)
   {
      d = c + 1;
      printf("% - 12.*s\n", d, string);
   }
   printf("\n");
   for(c = 11; c > = 0; c - -)
   {
      d = c + 1;
      printf("-12.*s\n", d, string);
   }
   printf("\n");
}
```

Summer 2017

1. What is an array? How to represent 2D-arrays in memory. [2 M]

Ans. Refer to Sections 5.1 and 5.3.

2. Explain the meaning of the following functions: [5 M]

(i) strstr

Ans. Refer Section 5.6.2.

(ii) strlen

Ans. Refer Section 5.6.2.

(iii) tolower

Ans. Refer Section 5.6.1.

(iv) strrev

Ans. Refer Section 5.6.2.

(v) strcpy

Ans. Refer Section 5.6.2.

3. Write a program to concatenate two strings. [5 M]

Ans. Refer to Program 5.33. [5 M]

4.

```
main()
{
   int m[ ] = {1, 2, 3, 4, 5}
   int x, y = 0;
   for (x = 0; x < 5; x++)
      y = y + m[x];
   printf("%d", y);
}
```

[5 M]

Ans. **Output:** 15

5.

```
main()
{
   char s1[ ] = "Kolkotta";
   char s2[ ] = "Pune";
   strcpy (s1, s2);
   printf("%s", s1)
}
```

Output: Pune

Summer 2018

1. Define string with example. 2 M

Ans. Refer to Section 5.5.

2. What are limitations of Array? 2 M

Ans. Refer to Section 5.1.

3. How can array be passed to function? Explain with example. 5 M

Ans. Refer to Section 5.4.

4. Write a C program to find the largest number from 3×3 matrix. 5 M

Ans. Refer to Chapter 5.

5. 5 M

```
main()
{
char * a[5] = {"good", "bad", "ugly", "wicked", "nice"};
printf("% s\n", a[0]);
printf("% s\n", * (a+2));
printf("% s\n", * (a[2]+2));
printf("% s\n", a[3]);
printf("% c\n", * (a[3]+2));
}
```

Ans. **Output:**

```
good
ugly
l
wicked
c
```

6. 5 M

```
main( )
{
static char str[ ] = "Malayalam";
char * s;
s = str + 8;
while s > = str)
{
printf("% c", * s);
s - -;
}
}
```

Ans. Error

6...

Functions

Learning Objectives...

- To understand the structure of function.
- To study, how C allows you to define functions according to your need.
- To learn various types of functions.
- To learn storage classes.

6.1 INTRODUCTION [W-14]

Functions are the building blocks of C language.

A number of statements grouped into a single logical unit are called a Function.

A Function is a group of statements that together perform a paticular task. Every C program has at least one function which is main(), and all the most trivial programs can define additional functions.

The use of function makes programming easier since repeated statements can be grouped into functions. Splitting the program into separate functions make the program more readable and maintainable.

C functions can be classified into two types:

1. Library Functions/Built in Functions, and
2. User-Defined Functions.

The Library Functions are the functions which are already defined in C's functions library i.e. header files.

For example, the functions scanf() and printf() are the library functions defined in file stdio.h same as functions sqrt() is defined in math.h and getch() is defined in conio.h.

User defined function is the function defined by the programmer who has written the program. The task to perform is decided by the user.

For example, the main() function is an user-defined function. We decide what is to be written in this function.

6.1.1 What is a Function/Meaning of Function

- Function is a set of instructions in a logical sequence, which performs specified task.

OR

- A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

OR

- The function is a self contained block of statements which performs a coherent task of a same kind.
- A program module or C function has following characteristics:
  1. It has unique name used to identify and invoke by function.
  2. List of parameters called arguments are passed to a function.
  3. Function body has program logic.
  4. Function returns some value in terms of some data type.
- If a program has many (multiple) functions, their definition may appear in any order. Because each function is independent of one another.
- A function will carry the specific action when we access it. The same function can be accessed from many places into the program. When the function action is finished, a program control will returned back to the next statement of a function call.

6.1.2 Purpose of Function

- Functions serve two purposes:
  2. They allow a programmer to say: `this piece of code does a specific job which stands by itself and should not be mixed up with anything else',
  3. They make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

Need of Functions:

- The functional program facilitates top-down modular programming approach as shown in Fig. 6.1.
- The length of source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function for further investigations.

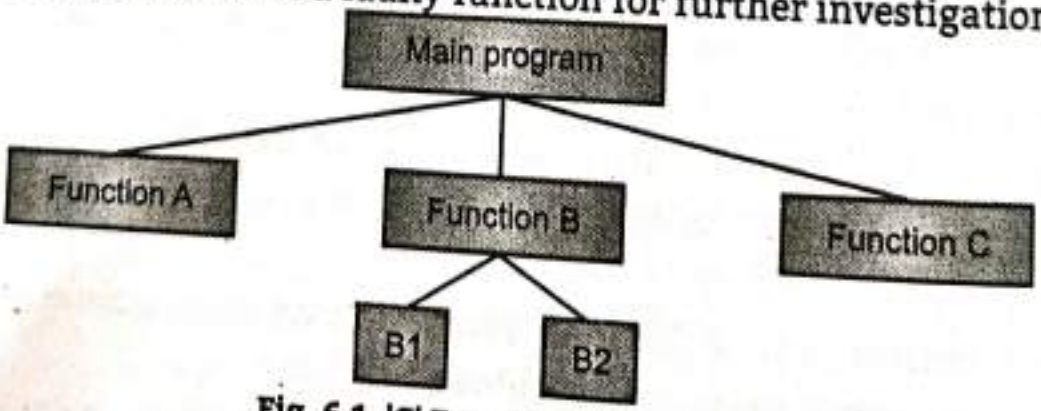
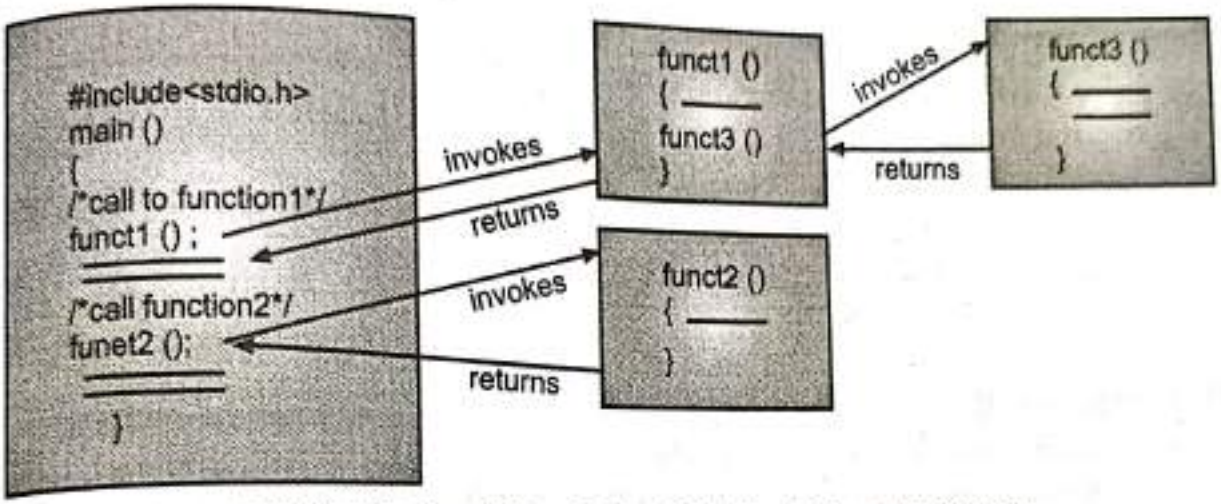


Fig. 6.1: 'C' Function Structure

- A function may be used by many other programs. This means that a C programmer can build on what others have already done. That is we can reuse the functions already defined in some program files.

6.1.3 How Function Works?

- A C language program does not execute the statements in a function until the function is invoked or called.
- When a C program function is called, control passes to the function and returns back to the calling part after the execution of function is over.
- Fig. 6.2 shows function calls and their returns.
- The calling program can send information to the function in the form of argument.
- An argument stores information or data needed by the function to perform its task.
- In C programming functions can send back information to the program in the form of a return value, (See Fig. 6.2).



main() calls funct1() and funct2(); funct1() calls funct3()

Fig. 6.2: Working of function

6.1.4 Advantages of Functions

- Various advantages of functions are:
  1. It is easy and simple to use.
  2. Debugging is more suitable for programs.
  3. It reduces the size of a program.
  4. It is easy to understand the actual logic of a program.
  5. Highly suited in case of large programs.
  6. By using functions in a program, it is possible to construct modular and structured programs.
  7. Using functions finding errors becomes easy and simple.

6.1.5 Function Definition

- The function definition is a separate program module that is specially written to implement the requirements of the function.
- So it is also called as function implementation. It includes following:
  - Function Header, and
  - Function Body

The **general form of function definition** is as given below:

```
return-type function-name(parameter list) //Function Header
   {
      local variable declarations
      statements                          // Function Body
      return value;
   }
```

- The first line function_type function_name(parameters list) is known as function header and the statements enclosing the curly braces are known as function body.

1. **Function Header: It includes three parts:**

   (i) **return type :**
   - Return Type is Type of value returned by function
   - Return Type may be "Void" if function is not going to return a value.

   (ii) **Function name:**
   - It is Unique Name that identifies function.
   - All Variable naming conversions are applicable for declaring valid function name.
   - The function name and the parameter list together constitute the function signature.

   (iii) **Parameters list:**
   - A parameter is like a placeholder.
   - When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
   - The parameter list refers to the type, order, and number of the parameters of a function.
   - Parameters are optional; that is, a function may contain no parameters.

For example:

```
int find_average(int x, int y int x)
{
--------
}
double square(double x, int n)
{
--------
}
float volume(int a, int b, int c)
{
--------
}
void print_area()
{
--------
}
```

2. **Function Body:** It contains the declarations and statements necessary for performing required task. The body enclosed in braces, contains three parts:
   (i) **Local variables declaration:** It specifies the variables needed by the function locally.
   (ii) **Function statements:** That actually performs task of the function.
   (iii) **The return statement:** It returns the value specified by the function.

For example:

```
float mul(float x, float y)
{
   float result; /* local variable */
   result = x * y; /* find the result */
   return(result); /* return the result */
}
void display(void)
{
   printf("Hello World!"); /* only print the value */
}
void sub(int a, int b)
{
   printf("Subtraction: %d", a - b); /* no variables */
   return; /* optional */
}
```

6.1.6 Function Declaration

- Like variables, all functions in C program must be declared, before they are used in calling function. This declaration of function is known as function prototype.
- It is having following syntax:

```
function_type function_name (parameter list);
```

this is very similar to function header except the terminating semicolon.

- For example, the function mul() will be declared as,

```
float mul(float, float);
```

- Generally, prototype declarations as not necessary, if the function have been declared before it is used. A prototype declaration may be placed in two places in the program.
  1. Above all the functions, and
  2. Inside function definition.
- We place declaration above all the functions this prototype is known as global prototype. That is, we can use this function in any part on the program.
- When we place the function definition in the local declaration section of another function it is referred as local prototype.

Comparison between Function Definition and Declaration:

Sr. No.	Function Definition	Function Declaration
1.	There is no semicolon at the end of the header.	There is a semicolon at the end of the declaration.
2.	The function body follows the header.	There is no function body.

6.1.7 Function Calls

(S-17)

- A function can be called by simply using function name followed by a list of actual parameters (or arguments) if any, enclosed in parentheses.
- If function is returning any value then we can store the return value in the variable which is of the same data type of that of return value.
- If the function does not return a value, then we declare it to be a function that returns void.
- Actual parameters are parameters as they appear in function calls.
- Formal parameters are parameters as they appear in function declarations.
- A parameter cannot be both a formal and an actual parameter, but both formal parameters and actual parameters can be either value parameters or variable parameters.
- The **general form of calling the function** is:

```
function name(list if arguments);
```

when we pass the variables to a function call, the arguments are called actual arguments.

- For example:

```
mul(5,3);
```

- Here, mul is the name of the function, and 5 & 3 are the actual values that we are passing to a function mul.

Example:

```
int mul(int,int);
void main()
{
   float m;
   m = mul(5, 3);    /* function call */
   printf("\n%d",m);
}
```

- When compiler encounters the function call, it transfers program control to the function mul(), by passing values 5 and 3 (actual parameters) to the formal parameters on the function mul().
- The mul() will performs operations on these values and then returns the result. It will be stored in variable m. We are printing the value of variable m on the screen.
- There are a number of different ways in which the function can be called as given below:

```
mul(5, 3);
mul(m, 3);
mul(5, m);
mul(m, n);
mul(m+5, 3);
mul(m+3, m+9);
mul(mul(6,8), 4);
```

- Only we have to remember that the function call must satisfy type and number of arguments passed as parameters to the called function.
- Function call is of two types:
  1. **call by value:** When we call function by passing normal values or variables, the function call is called as call by value.
  2. **call by reference:** At the time of function call instead of passing variables or values if we pass the reference of variables then such kind of function call is called as call by reference.

Return Values:

- A function may or may not send back any value to the calling function. If it does, it is done by return statement.
- The return statement also sends the program control back to the calling function.
- It is possible to pass a calling function any number of values but called function can only return one value per call.
- The return statement can take one of the following forms (syntax):

```
return;
   OR
return(value);
   OR
return(variable);
   OR
return(expression);
```

- The first plain return does not return any value; it acts much as closing brace of the function. The remaining three statements can eliminates the brackets also. When the return is executed, the program control immediately transfers back to the calling function. None of the statements written after return are executed afterwards.
- For example:

```
return(x);
printf("Bye...Bye");
```

- In this case the printf statement will not be executed in any case. Because the return statement will transfer program control immediately back to the calling function.
- Some examples of return are:

```
int div(int x, int y)
{
   int z;
   z = x / y;
   return z;
}
```

- Here, instead of writing three statements inside the function div(), we can write a single statement as,

```
return(x*y);
  OR
return x*y;
```

- A function may have more than one return statement when it is associated with any condition such as,

```
if(x>y)
return x;
else
return y;
```

In this code, in any condition, only one return statement will be executed.

6.2 TYPES OF FUNCTIONS [S-18]

- There are two types of functions as:
  1. Built in functions or Pre-defined functions, and
  2. User defined functions.

1. Built in Functions:

- These functions are also called as 'library functions'.
- These functions are provided by system.
- These functions are pre-written, compiled and placed in libraries.
- Various stdio.h library functions are:

Function Name	Prototype	Purpose
1. getchar()	int getchar(void)	gets a character from stdin.
2. putchar()	int putchar(int c)	writes a character to stdout.
3. gets()	char *gets(char *)	gets a string from stdio.
4. puts()	int puts(const char*)	outputs a string to stdout.
5. printf()	int printf(const char* format, [arg, ...]);	writes a character to stdout.
6. scanf()	int scanf(const char* forma, [address, ...]);	scans and formats an input from stdin.
7. sprintf()	int sprintf(char* buffer, char* format, [argument, ...]);	writes formatted output to a string.
8. sscanf()	int sscanf(const char* buffer, const char* format, [address, ...]);	scans and formats input from a string.
9. fflush()	int fflush(file *);	flushes a stream.

- Various math.h functions are:

Function Name	Prototype	Purpose
1. abs()	int abs(int x)	Returns the absolute value of x.
2. cos()	double cos(double x)	Returns cosine of x.
3. exp()	double exp(double x)	Calculates ex.
4. floor()	double floor(double x)	Returns the largest integer<=x.
5. log()	double log(double x)	Returns natural log of x.
6. pow()	double pow(double x, double y)	Calculates xy.
7. sin()	double sin(double x)	Calculated sine of x.
8. sqrt()	double sqrt(double x)	Calculates square root of x.

- Various conio.h functions are:

Function Name	Prototype	Purpose
1. clrscr()	void clrscr(void)	Clears the text mode window.
2. clreof()	void clreof(void)	Clears to end of line in text window.
3. getch()	int getch(void)	Gets a character from console.
4. getche()	int getche(void)	Same as getch but echoes to screen. No buffering is done.
5. kbhit()	int kbhit(void)	Returns an integer corresponding to a keystroke.
6. putch()	int putch(int ch)	Outputs a character to the text window on screen.

- Various stdlib.h functions are:

Function Name	Prototype	Purpose
1. atof()	double atof(const char *s)	Converts a string to float.
2. atoi()	double atoi(const char *s)	Converts a string to int.
3. atoll()	double atoll(const char *s)	Converts a string to long.
4. random()	int random(int num)	Returns an integer between 0 and (num-1).
5. randomize()	void randomized(void)	Initialize the random number generator with a random value.
6. system()	int system(const char* command)	Used to execute an MS-DOS command.

2. User Defined Functions:

- The functions which are created by user for program are known as 'User defined functions'.

Syntax:

```
void main()
{
   // Function prototype
   <return_type><function_name>([<argu_list>]);
   // Function Call
   <function_name>([<arguments>]);
}
// Function definition
<return_type><function_name>([<argu_list>]);
{
   <function_body>;
}
```

Advantages of user defined functions:

- User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.

- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can divide the workload by making different functions.

Program 6.1: Program to demonstrate addition of two numbers using function.

```
#include<stdio.h>
#include<conio.h>
void add()
{
    int a, b, c;
    clrscr();
    printf("\n Enter Any 2 Numbers: ");
    scanf("%d %d",&a,&b);
    c = a + b;
    printf("\n Addition is: %d",c);
}
void main()
{
    void add();
    add();
    getch();
}
```

Output:

```
Enter Any 2 Numbers: 23 6
Addition is: 29
```

Categories of the Functions:

- Depending upon whether arguments are present or not and whether value is returned or not functions are categorized as:
  1. Category 1: Functions with no arguments and no return value
  2. Category 2: Functions with arguments and no return value
  3. Category 3: Functions with no arguments and a return value
  4. Category 4: Functions with arguments and a return value

1. Functions with No Arguments and No Return Value:

- In this type of function, the main program will not send any arguments to the calling function and also the function will not return any value to the main program.
- The general form given below is:

```
return-type function-name (formal parameter type list);
```

- Let's see some examples for this category where no arguments will be passed and no value will be returned.

Program 6.2: Write a program to demonstrate the function with no arguments and no return value.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    clrscr( );
    printf("\n You are welcome to the main program");
    message( );
    printf("\n Welcome back to the main");
}
message( )
{
   printf("\n Welcome to the subprogram");
}
```

Output

```
You are welcome to the main program
Welcome to the subprogram
Welcome back to the main
```

Program 6.3: Write a program to calculate the area of circle.

```
#include<stdio.h>
#include<conio.h>
 void area();  // Prototype Declaration
void main()
{
    clrscr();
    area();
    getch();
}
void area()
{
   float ar; rad;
   printf("\nEnter the radius : ");
   scanf("%f",&rad);
   ar = 3.14 * rad * rad ;
   printf("Area of Circle = %f",ar);
}
```

Output

```
Enter the radius: 3
Area of Circle = 28.260000
```

Program 6.4: Write a program to print even numbers up to 30.

```
#include<stdio.h>
#include<conio.h>
void even();  // Prototype Declaration
void main()
{
    clrscr();
    even();
    getch();
}
void even()
{
   int i;
   printf("\n Even numbers from 1 to 30 are:");
   for(i=1;i<=30;i++)
   {
          if(i%2==0)
      {
        printf("\t%d",i);
       }
   }
}
```

Output:

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

2. Functions with Arguments and No Return Value:

- In this type of function, Function accepts argument but it does not return a value back to the calling Program. It is Single (One-way) Type Communication.
- The **general form** for a function declaration:

```
return-type function-name (formal parameter type list);
```

- A void functions with value parameters are declared by enclosing the list of types for the parameter list in the parentheses.
- In this type of category of functions, we pass arguments as an actual arguments and the called function receive these values in its definition as formal parameters. But nothing will be returned by the called function to the calling function.
- The number of actual parameter should be equal to the formal parameters and also the data type of both the parameter should be same.
- Let's see some examples for this category where arguments will be passed and no value will be returned.

Program 6.5: Write a program to demonstrate the functions with arguments and no return value.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
printf("\n Enter a value for a :");
```

```
scanf("%d",&a);
printf("\n Enter a value for b :");
scanf("%d",&b);
funct(a,b);
getch();
}
funct(int a1, int b1)
{
int c;
c=a1+b1;
printf("\n Result = %d",c);
return 0;
}
```

Output:

```
Enter a value for a : 12
Enter a value for b : 10
Result = 22
```

Program 6.6: Write a program to find the average of the three numbers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
printf("\n Enter a value for a :");
scanf("%d",&a);
printf("\n Enter a value for b :");
scanf("%d",&b);
printf("\n Enter a value for c :");
scanf("%d",&c);
average(a,b,c);
getch();
}
average(int a1, int b1, int c1)
{
int avg;
avg=(a1+b1+c1)/3;
printf("\n Average is  = %d",avg);
return 0;
}
```

Output:

```
Enter a value for a : 12
Enter a value for b : 10
Enter a value for c : 8
Average is = 10
```

Program 6.7: Write a program to print all prime numbers up to the number entered by user.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int n;
   printf("\n Enter the limit :");
   scanf("%d",&n);
   prime(n);
   getch()
}
void prime(int limit)
{
int i,j,flag;
for(i=2;i<=limit;i++)
{
   flag=0;
   for(j=2;j<=i/2;j++)
   {
   if(i%j==0)
      {
      flag=1;
         break;
      }
      if(flag==0)
      printf("\t%d",i);
   }
}
}
```

Output:

```
Enter the limit: 10
2  3  5  7
```

3. Functions with no Arguments and a Return Value:

- In this type of function, the calling program will not send any arguments to the called function but the called function will return the value to the calling function.
- The **general form** is given below:

```
return-type function-name (formal parameter type list);
```

- Let's see some examples for this category where no arguments will be passed and value will be returned.

Program 6.8: Write a program to demonstrate the Functions with no arguments but returns a value.

```
#include<stdio.h>
#include<conio.h>
int send()
{
   int no;
   printf("\n Enter a number : ");
   scanf("%d",&no);
   return(no);
}
```

```
void main()
{
    int z;
    clrscr();
    z = send();
    printf("\nYou entered : %d.", z);
    getch();
}
```

Output:

```
Enter a number : 25
You entered : 25
```

Program 6.9: Write a program to add digits of a number entered by user.

```
#include<stdio.h>
#include<conio.h>
int sum()
{
    int no, res=0, remainder;
    printf("\n Enter a number : ");
    scanf("%d",&no);
    while(no != 0)
   {
      remainder = no % 10;
      res = res + remainder;
      no = no / 10;
   }
    return(res);
}
 void main()
{
    int z;
    clrscr();
    z = sum();
    printf("\n The sum of  the digits is : %d.", z);
    getch();
}
```

Output:

```
Enter a number : 1234
The sum of the digits is : 10
```

4. Functions with arguments and a Return Value:

- This type of function can send arguments (data) from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function.
- This type of function is mostly used in programming world because it can do two way communications.
- It can accept data as arguments as well as can send back data as return value.

- Let's see some examples for this category where arguments will be passed and value will be returned.

Program 6.10: Program to demonstrate the Functions with arguments and returns a value.

```
#include<stdio.h>
#include<conio.h>
 int add(int x, int y)
{
    int result;
    result = x+y;
    return(result);
}
void main()
{
    int z;
    clrscr();
     z = add(952,321);
    printf("Result %d.\n\n",add(30,55));
    printf("Result %d.\n\n",z);
     getch();
}
```

Output:

```
Result 85
Result 1273
```

Program 6.11: Program to find HCF (highest common factor) and LCM (least common multiple).

```
#include <stdio.h>
#include <conio.h>
long gcd(long, long);
void main()
{
  long x, y, hcf, lcm;
  clrscr();
  printf("Enter two integers\n");
  scanf("%ld%ld", &x, &y);
  hcf = gcd(x, y);
  lcm = (x*y)/hcf;
  printf("Greatest common divisor of %ld and %ld = %ld\n", x, y, hcf);
  printf("Least common multiple of %ld and %ld = %ld\n", x, y, lcm);
  getch();
}
long gcd(long x, long y)
 {
  if (x == 0)
 {
    return y;
 }
```

```
while (y != 0)
{
   if (x > y)
   {
      x = x - y;
   }
   else
   {
      y = y - x;
   }
  }
   return x;
}
```

Output:

```
Enter two integers
9 15
Greatest common divisor of 9 and 15 = 3
Least common multiple of 9 and 15 = 45
```

Program 6.12: Program to find factorial of a number.

```
#include <stdio.h>
#include<conio.h>
long factorial(int);
void main()
{
  int number;
  long fact;
  clrscr();
  printf("\n Enter a number to calculate it's factorial : ");
  scanf("%d", &number);
  printf("\n The factorial of a number is :%d", factorial(number));
  getch();
}
 long factorial(int n)
{
  int j;
  long result = 1;
  for (j = 1; j <= n; j++)
  result = result * j;
  return (result);
}
```

Output:

```
Enter a number to calculate it's factorial : 5
The factorial of a number is : 120
```

6.3 CALL BY VALUE AND CALL BY REFERENCE [W-14]

6.3.1 Call by Value

- The transfer of values between the actual and formal parameter takes place by call by value.
- In this case, the values get passed through actual parameter to formal parameter when a function gets called. In this case only one value can be returned.
- When a function is called by value of variables then that function is known as 'function call by values.'
- Function in C passes all arguments by value. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function.
- Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value.

Syntax:

```
// Declaration
void <function_name>(<data_type><var_nm>);
// Calls
<function_name>(<var_nm>);
// Definition
void <function_name>(<data_type><var_nm>);
{
   <function_body>;
   - - - - - - - -;
}
```

Program 6.13: Program for call by value.

```
#include<stdio.h>
#include<conio.h>
int add(int p,int q);
int main()
{
   int a,b,c;
   clrscr();
   printf("Enter two numbers \n");
   scanf("%d %d", &a, &b);
   c = add(a,b);
   printf("\nSum of %d and %d is %d", a, b, c);
   getch();
   return 0;
}
```

```
int add(int p, int q)
{
   int result;
   result=p+q;
   return(result);
}
```

Output:

```
Enter two numbers
10
20
Sum of 10 20 is 30
```

6.3.2 Call by Reference

- In call by reference method, the address of an argument copied into the parameter.
- Call by reference can be achieved by passing a pointer to an argument. As we pass address of the argument, the code within the function can change the value of the arguments outside the function.
- Pass by Reference mechanism is used when you want a function to do the changes in passed parameters and reflect those changes back to the calling function. In this case only addresses of the variables are passed to a function so that function can work directly over the addresses.

Program 6.14: Program for call by reference.

```
#include<stdio.h>
#include<conio.h>
void add(int *p, int *q, int *r);
void main()
{
   int a,b,c;
   clrscr();
   printf("Enter two numbers \n");
   scanf("%d %d", &a, &b);
   add(&a, &b, &c);
   printf("\nSum of %d and %d is %d", a, b, c);
   getch();
}
void add(int *p, int *q, int *r)
{
   *r = *p + *q;
}
```

Output:

```
Enter two numbers
10
20
Sum of 10 20 is 30
```

Difference between call by value and call by reference:

Call by value	Call by reference
1. This is the usual method to call a function in which only the value of the variable is passed as an argument.	1. In this method, the address of the variable is passed as an argument.
2. Any alternation in the value of the argument passed is local to the function and is not accepted in the calling program.	2. Any alternation in the value of the argument passed is accepted in the calling program(since alternation is made indirectly in the memory location using the pointer).
3. Memory location occupied by formal and actual arguments is different.	3. Memory location occupied by formal and actual arguments is same and there is a saving of memory location.
4. Since a new location is created, this method is slow.	4. Since the existing memory location is used through its address, this method is fast.
5. There is no possibility of wrong data manipulation since the arguments are directly used in an application.	5. There is a possibility of wrong data manipulation since the addresses are used in an expression. Here a good skill of programming is required.

6.4 STORAGE CLASSES [S-17]

- 'Storage' refers to the scope of a variable and memory allocated by compiler to store that variable.
- Scope of a variable is the boundary within which a variable can be used.
- Storage class defines the scope and lifetime of a variable.
- We can define scope of a variable as, the region of the program in which the variable is valid or visible.
- Storage class refers to the manner in which memory is allocated by the compile to variables.
- A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.
- It determines the scope and lifetime of a variable. There are two types of locations in a computer where such a value stored one is Memory and another is CPU registers.
- There are four types of storage classes:
  1. Automatic,
  2. Register,
  3. Static, and
  4. External.
- The storage class of a variable answers the following questions:
  1. Where the variable would be stored?
  2. What will be the default initial value?
  3. What is the scope of the variable?
  4. How long would the variable exist?

6.4.1 Auto Storage Class

- These classes are created when the function is called and destroyed automatically when the function is exited.
- Auto storage class is the default for local variables.
- **Syntax:** `auto [data_type] [variable_name];`

Example:

```
int num(int n)
{
   int count;
   auto int Employee;
}
```

- Above example defines two variables with the same storage class auto class can only be used within functions, i.e. local variables.
- The auto variables are created when the function is called and declared automatically when the function is exited.
- The variable in C can have any one of the four storage classes.
- **Automatic Variables:** They are declared inside a function. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic variables are also referred to as local or internal variables. When a variable is declared inside a function without storage class, by default, it is an automatic variable. The keyword 'auto' is used to declare automatic variables.
- One important feature of automatic variable is that their value cannot be changed accidentally.

Features of Auto Storage Classes:

1.	Scope	: Local to the block in which it is defined.
2.	Default initial value	: Garbage value
3.	Life	: Till the control remains within the block where it is defined
4.	Storage	: Memory

Program 6.15: Program to demonstrate automatic storage class.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   auto int i=10;
   clrscr();
   {
      auto int i=20;
      printf("\n\t %d",i);
   }
   printf("\n\n\t %d",i);
   getch();
}
```

Output:

```
20
10
```

6.4.2 Extern Storage Class

- Extern is used to give a reference of a global variable that is visible to ALL the program files.
- When you use 'extern' the variable cannot be initalized as all it does is point the variable name at a storage location that has been previously defined.
- Extern storage class defines a global variable that is accessible by any function in the program. This class are declared outside all functions.

  Syntax: `extern [data_type] [variable_name];`

  Example: `extern int a;`
- The variable access time is very fast as compared to other storage classes. But few registers are available for user programs.
- The variables of this class can be referred to as 'global or external variables.' They are declared outside the functions and can be invoked at anywhere in a program.
- **External variables:** Variables that are both alive and active throughout the entire program are known as external variables. They are also known as Global variables. These variables can be accessed by any function in the program. They are declared outside the function. The keyword 'extern' can be used to declare a variable inside the function main.
- In some cases, the same global variable defined in file X is used again in file Y. Then how does the compiler know that the variable used in file Y is actually the same variable declared in file X.
- The answer is, use the extern specifier to the global variable defined elsewhere. In this condition, we declare a global variable in file X, and then declare the variable again using the extern specifier in file Y.
- For example, suppose we have two global int variables, a and b, that are defined in one file, and then in another file, you may have the following declarations.

```
int a = 0;          /* a is a global variable */
extern int b;       /* an allusion to a global variable y */
int main( )
{
   extern int c;    /* an allusion to a global variable z */
   int I;           /* I is a local variable */
   .
   .
   return 0;
}
```

Features of Extern – storage class are listed below:

1. Scope : Global
2. Storage : Memory
3. Life : Active throughout the entire program
4. Default initial value : Zero

Program 6.16: Program to demonstrate external storage class.

```
#include<stdio.h>
#include<conio.h>
extern int i=10;
void main()
{
   int i=20;
   void show(void);
   clrscr();
   printf("\n\t %d",i);
   show();
   getch();
}
void show(void)
{
   printf("\n\n\t %d",i);
}
```

Output:

```
20
10
```

6.4.3 Static Storage Class

- Static storage class is the default for global variables. The two variables i.e. count and road in the following example have a static storage class.
- Static storage class can be used only if we want the value of a variable to persist between different function calls.

Syntax: `static [data_type] [variable_name];`

For example,

```
data int count;
int road;
main( )
   {
      printf("%d\n", count);
      printf("%d\n", road);
   }
```

- **Static variables:** The value of static variables persists until the end of the program. A variable can be declared static using the keyword 'static' like static int x. A static variable may be either an internal type or an external type, depending on the place of declaration. Internal static variable can be used to retain values between function calls. Static variable is initialized only once, when program is compiled.
- Static storage class can also be defined to local variables and they retain their values between calls to the function.

For example:

```
void Func(void)
{
    static count=1;
}
```

Features of static storage class for listed below:

1. Scope : Local to the block in which it is defined
2. Default initial value : Zero
3. Storage : Memory
4. Life : Persists between different function calls.

Program 6.17: Program to demonstrate static storage class.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    void incre(void);
    clrscr();
    for (i=0; i<3; i++)
    incre();
    getch();
}
void incre(void)
{
    int avar=1;
    static int svar=1;
    avar++;
    svar++;
    printf("\n\n Automatic variable value: %d",avar);
    printf("\t Static variable value: %d",svar);
}
```

Output:

```
Automatic variable value: 2   Static variable value: 2
Automatic variable value: 2   Static variable value: 3
Automatic variable value: 2   Static variable value: 4_
```

6.4.4 Register Storage Class

- Register storage class is used to define local variables that should be stored in a CPU register instead of memory i.e. the variable has a maximum equal to the register size and can't have the unary '&' operates applied to it.

  Syntax: `register [data_type] [variable_name];`
- Register is used to define local variables that should be stored in a register instead of RAM.

Register variables: We can tell the compiler that a variable should be kept in one of the machine's register, instead of keeping in memory. Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs. These variables can be declared as,

```
register int count;
```

- Since, only a few variables can be placed in the register, it is important to carefully select the variables for this purpose.
- Once limit is reached, C will automatically convert register variables into non-register variables.

For example:

```
{
   register int items;
}
```

Features of Register-storage class are given below:

1. Scope : Local to the block in which it is defined
2. Storage : CPU registers
3. Life : Till the control remains within the block where it is defined
4. Default initial value : Garbage value.

Program 6.18: Program to demonstrate register storage class.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   register int i=10;
   clrscr();
   {
      register int i=20;
      printf("\n\t %d",i);
   }
   printf("\n\n\t %d",i);
   getch();
}
```

Output:

```
20
10
```

Comparison of Storage Classes:

Features	Automatic Storage Class	Register Storage Class	Static Storage Class	External Storage Class
Keyword	auto	register	static	extern
Initial Value	Garbage	Garbage	Zero	Zero
Storage	Memory	CPU register	Memory	Memory

contd. ...

Scope	scope limited, local to block	scope limited, local to block	scope limited, local to block	Global
Life	limited life of block, where defined	limited life of block, where defined	value of variable persist between different function calls	Global, till the program execution
Memory location	Stack	Register memory	Segment	Segment

Example	`void main()`
	`{`
	`auto int i;`
	`printf("%d",i);`
	`}`
	Output:
	124
		`void main()`
		`{`
		`register int i;`
		`for(i=1; i<=5 ; i++);`
		`printf("%d ",i);`
		`}`
		Output:
		1 2 3 4 5
			`void add();`
			`void main()`
			`{`
			`add();`
			`add();`
			`}`
			`void add()`
			`{`
			`static int i=1;`
			`printf("\n%d",i);`
			`i=i+1;`
			`}`
			Output:
			1
			2
				`void main()`
				`{`
				`extern int i;`
				`printf("%d",i);`
				`int i=5`
				`}`
				Output:
				0

Summary

- To avoid repetition of code and bulky programs, functionally related statements are isolated into a function.
- Function declaration specifies what is the return type of the function and the types of parameters it accepts.
- Function definition defines the body of the function.
- Variables declared in a function are not available to other functions in a program. So, there won't be any clash even if we give same name to the variables declared in different functions.
- A function can be called either by value or by reference.
- Register storage class is used to define local variables that should be stored in a CPU register instead of memory
- The scope of a static variable is local to the function in which it is defined but it doesn't get terminated when the function execution gets over.
- Variables having extern storage class have a global scope. Extern variables are used when programmers want a variable to be visible outside the file in which it is declared.
- Auto storage class applied to local variables and those variables are visible only within the function inside which it is declared and it gets terminated as soon as the function execution gets over.

Check Your Understanding

1. What is the default return type if it is not specified in function definition?
   (a) void (b) int
   (c) float (d) short int
2. The default parameter passing mechanism is _____
   (a) Call by value. (b) call by reference.
   (c) call by value result. d) None.
3. Pick the correct statements.
   (I) The body of a function should have only one return statement.
   (II) The body of a function may have many return statements.
   (III) A function can return only one value to the calling environment.
   (IV) If return statement is omitted, then the function does its job but returns no value to the calling environment.
   (a) (I) and (II) (b) (I) and (III)
   (c) (II) and (III) (d) (II) and (IV)
4. Functions can return structure in c?
   (a) True (b) False
5. In C, what is the meaning of following function prototype with empty parameter list

```
void fun()
{
/* .... */
}
```

   (a) Function can only be called without any parameter.
   (b) Function can be called with any number of parameters of any types.
   (c) Function can be called with any number of integer parameters.
   (d) Function can be called with one integer parameter.

Answers

1. (b)	2. (a)	3. (c)	4. (a)	5. (b)

Trace the Output

(a)
```
main( )
{
   printf ( "\nOnly programmer use C?" ) ;
   display( ) ;
}
display( )
{
   printf ( "\nStudent too use C!" ) ;
   main( ) ;
}
```

(b)
```
main( )
{
    printf ( "\nC to it that C survives" ) ;
    main( ) ;
}
```

(c)
```
main( )
{
    int i = 45, c ;
    c = check ( i ) ;
    printf ( "\n%d", c ) ;
}
check ( int ch )
{
    if ( ch >= 45 )
    return ( 100 ) ;
    else
    return ( 10 * 10 ) ;
}
```

(d)
```
main( )
{
    int i = 45, c ;
    c = multiply ( i * 1000 ) ;
    printf ( "\n%d", c ) ;
}
check ( int ch )
{
    if ( ch >= 40000 )
    return ( ch / 10 ) ;
    else
    return ( 10 ) ;
}
```

(e)
```
main( )
{
    float a = 15.5 ;
    char ch = 'C' ;
    printit ( a, ch ) ;
}
printit ( a, ch )
{
    printf ( "\n%f %c", a, ch ) ;
}
```

Practice Questions

Q.1 Write the answer of following questions in brief.

1. How to declare a function?
2. What is meant by storage classes?
3. Type of functions?
4. What is user defined function.
5. List the categories of function

Q.2 Write the answer of following questions .

1. State advantages and limitations of function.
2. Explain the following parameters passing methods.
   (a) Call by value, (b) Call by reference.
3. What is a function? How are the functions declared?
4. Write a function to add all even numbers together and add all odd numbers together and print the even sum and odd sum. Accept the number range from the user.
5. Write a function to find out the GCD of 2 given integers.

Q.3 Define terms:

1. Function, 2. Storage classes, 3. Function with parameter and return value, 4. Static storage, 5. Auto storage, 6. Register storage, 7. Extern storage, 8. Call by value, 9. Call by reference

Previous Exam Questions

Winter 2014

1. Define function and explain function declaration, function definition and function call with example. [5 M]

Ans. Refer to Sections 6.1 and 6.3.

Summer 2015

1. [5 M]

```
main()
{
   int i=-5,j=-2;
      junk(i,&j);
   printf("i=%d j=%d\n",i,j);
   junk(int i, int*j)
   {
      i=i*j;
      *j=*j**j;
   }
}
```

Ans. **Output:** Shows syntax error regarding 'use of pointer'.

Winter 2015

1. [5 M]

```
int test(int number)
{
   int m, n = 0;
   while (number)
   {
      m = number%10;
      if (m%2)
         n = n+1;
      number = number | 10;
   }
      return(n);
```

What will be the values of x and y when following statements are executed?

```
int x = test (135);
int y = test (246);
```

Output:

```
values of x = 3
values of y = 0
```

Summer 2016

1. Trace output [5 M]

```
int prod (int m, int n);
main()
{
   int X = 10;
   int Y = 20;
   int p, q;
   p = prod(X, Y)
   q = prod(p, prod (X, Z));
   printf("%d%d, \n", p, q);
}
int prod (int a, int b)
{
   return (a * b);
}
```

Winter 2016

1. Define function. [2 M]

Ans. Refer to Section 6.1.

2. Trace output [5 M]

```
void test (int *a);
main( )
{
   int X = 50;
   test (&X);
   printf ("%d\n", X);
}
Void test (int *a);
{
   *a = *a + 50;
}
```

Output: 100

Summer 2017

1. What is formal and actual parameter? [2 M]

Ans. Refer to Section 6.1.8.

2. Explain the different storage classes in C. [5 M]

Ans. Refer Section 6.4.

7... Introduction to Pointer

Learning Objectives...

- To study concept of Pointer.
- To understand need of allocation memory.
- To understand, how to pass arguments to function.
- To learn Array representation and passing of arrays.

7.1 INTRODUCTION TO POINTERS [W-14, 15]

- Pointers is a variable that stores the address of another variable.
- A Pointer in C is used to allocate memory dynamically i.e. at run time.
- The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.
- For example, if one variable contain the address of another variable, the first variable is said to point the second variable.
- Pointers are used in C program to access the memory and manipulate the address. Fig. 7.1 shows the above situation.

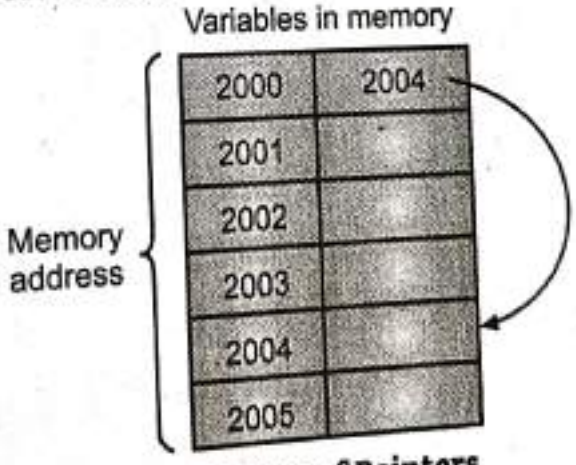


Fig. 7.1: Use of Pointers

- A pointer is a variable that represents the address (location) of a data item in the computer's memory.
- Every data item is stored in memory, occupies a space somewhere in memory.
- Pointers are represented by '*'. It is a derived data type in C.

(7.1)

- Pointer returns the value of stored address.

  Syntax: `<data_type> * pointer_name;`

  Example: `int *p; char *p;`

  Where, * is used to denote that "p" is pointer variable and not a normal variable.

7.1.1 Definition of Pointer

- A pointer is a variable that represents the location (rather than the value) of a data item such as a variable or an array element.

OR

- Pointers are the variables that contain the address of another variable within the memory.

7.1.2 Features of Pointer

- Features of pointer are listed below:
  1. Pointer variable should have prefix '*'.
  2. Combination of data types is not allowed.
  3. Pointers are more effective and useful in handling arrays.
  4. It can also be used to return multiple values from a function using function arguments.
  5. It supports dynamic memory management.
  6. It reduces complexity and length of a program.
  7. It helps to improve execution speed that results in reducing program execution time.

7.1.3 Need of Pointers

- Following points describes need of pointers:
  1. Pointer is used for creating data structures such as linked lists, trees, graphs etc.
  2. Pointer enhances the capability of the language to manipulate data.
  3. Pointers allows working with dynamically allocated memory.
  4. Pointers reduce the length and complexity of the program.
  5. Pointers increases the execution speed.

7.1.4 Advantages of Pointers

- Pointer consists of the following advantages:
  1. Pointers provide an alternate way to access individual array element.
  2. Pointers provide a convenient way to represent multi-dimensional arrays, allowing a single multidimensional array to be replaced by a lower-dimensional array of pointers.
  3. Pointers are more efficient in handling complex data structures and data tables.
  4. Pointers reduce the length of a program.
  5. Pointers increase the program execution speed.

7.1.5 Applications of Pointers

Applications of pointers are listed below:

1. Pointers can be used to simulate passing parameters by reference i.e. the arguments can be modified.
2. They provide an alternate method to access array elements.
3. They are used for passing arrays and strings to functions.
4. They are more efficient in handling complex data structures like linked lists, trees, graphs, etc.
5. One of the most important use of pointer is a dynamic memory allocation where memory is allocated and released for a variable during run-time.

7.1.6 Declaration of Pointer

[S-16]

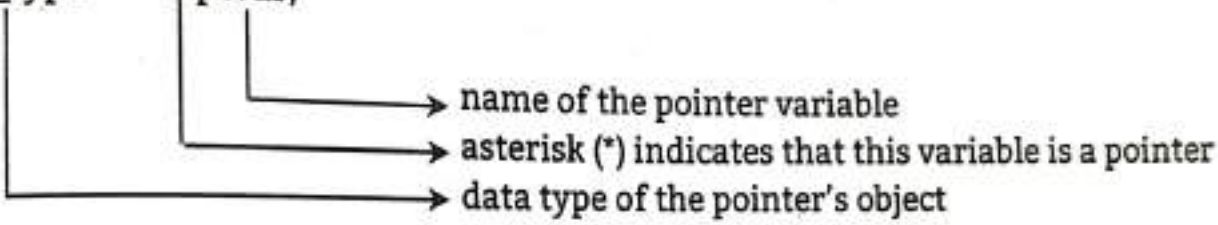
Pointer variables must be declared before they may be used in a C program.

When a pointer variable is declared, the variable name must be proceeded by an asterisk (*) sign.

This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the object of the pointer.

Syntax for pointer are given below:

```
data_type  * ptvar;
```



Above syntax tells the compiler that:

1. ptvar is a pointer.
2. ptvar needs memory location.
3. ptvar points to a variable of type data_type.

OR

```
data_type * pointer_name;
```

where, data types is C datatype and pointer_name is name of pointer which is a valid C identifiers.

Example:

```
int *p;          // declares that 'p' is a pointer that points to an integer data type.
float *xyz;      // declares that 'xyz' is a pointer that points to a floating point variable.
char *pragati;   // declares that 'status' is a pointer that points to a char variable.
```

[S-16]

7.1.7 Initialisation of Pointer

The process of assigning the address of a variable to a pointer variable is known as initialization.

After declaration, we have to initialize the pointer variable. Use of the 'address of (&)' operator as a prefix to the variable name assigns its address to the pointer.

- **General form of pointer initialization** is given below:

```
pointer=&variable;
```

Example:

```
zptr = &zvar;     // assigns the address of variable 'zvar' to pointer 'zptr'
```

- We can also assign a pointer value to another variable of the same type.

Example:

```
int z = 5, x, *zptr;
zptr = &z;    // assigns the address of variable 'z' to the pointer 'zptr'
x = *zptr;    // assigns the value at the address pointed by 'zptr' to variable x i.e. x = 5
zptr = 0      // assigns value 0 to 'z' since z = *zptr
```

The two statements,

```
zptr = &z;
x = *zptr;
```

are equivalent to single statement ,

```
x = *(&z);
```

or

```
x = z;
```

- That is, the & operator is the inverse of *operator.

Program 7.1: Program which demonstrates the declaration of pointer.

```
#include<stdio.h>
void main( )
{
   int l = 2;
   int * m;
   m = &l;
   printf("\n Address of l = %u", m);
   printf("\n Value of l = %d", * m);
}
```

Output:

```
Address of l = 65524
Value of l = 2
```

7.2 INDIRECTION OPERATOR AND ADDRESS OF OPERATOR [S-15]

1. Indirection Operator (*):

- The value of variable using pointer can be accessed using unary operator * (asterisk), known as the indirection operator or dereferencing operator.
- De-referencing is the operation performed to access data contained in the memory location pointed by the pointer.

For example:

```
int d1, d2, *p;
d1=30;
p=&d1;
d2=*p;
```

In above statements d1, d2 are integer variables and p is a integer pointer variable. The value 30 is stored in variable d1. Then the address of variable d1 is stored into p. The *p returns the value of variable d1, which is stored into variable d2. That means the above statements copies the value of variable d1 into variable d2.

The statements,

```
p=&d1;
d2=*p;
```

are equivalent to the following single statement,

```
d2=* &d1;
```

which is equivalent to,

```
d2 = d1;
```

- The unary operator '&' gives the address of a variable.
- The unary operator '*' gives the value of address.

Program 7.2: Program to illustrate the use if dereferencing or indirection operator '*'.

```
#include<stdio.h>
void main()
{
   int d1=30, *p;
   p=&d1;
   printf("\nThe value of d1=%d",d1);
   printf("\nThe address of d1=%u",p);
   printf("\nThe value of d1=%d",*p);
   printf("\nThe value of d1=%d", * &d1);
   return 0;
}
```

Output:

```
The value of d1=30
The address of d1=65524
The value of d1=30
The value of d1=30
```

2. Address of Operator (&):

- The address operator (&) must act upon operands that associated with unique address, such as ordinary variable or single array element. Thus the address operators cannot act upon arithmetic expressions.

Program 7.3: Program for & operator.

```
#include <stdio.h>
int main()
{
int *ptr, q;
 q = 50;
 /* address of q is assigned to ptr */
ptr = &q;
 /* display q's value using ptr variable */
printf("%d", *ptr);
return 0;
}
```

Output:

```
50
```

7.3 POINTER ARITHMETIC

- Number of operations can be performed on pointers since they are variables. Here, we see the following operations perform on a pointers.
  1. Increment-decrement.
  2. Adding a number from a pointer.
  3. Subtracting a number from a pointer.
  4. Subtraction of one pointer from another.
  5. Comparison of two pointer variables.
- Some operations would give unpredictable results if performed.

1. Increment-Decrement:

- Suppose pt is a pointer pointing to integer type of data. i.e.

```
int * pt;
```

- If we give pt++ then it increments pt to point to next location of same type. What do we mean by same type? If pt is a pointer to integer, integers occupy 2 bytes. So if pt = 4001, pt++ gives pt = 4003. If it points char data then it will move 1 bytes ahead. pt--decrements pt to point to previous element of the same type.
- Suppose char type data need 1 byte and int type data need 2 bytes. So when a char pointer is incremented its value increase by 1 and when a int pointer is incremented its value increases by 2 or the increment will take according to the base type it points. One can even add or subtract integers from pointer values.

For example:

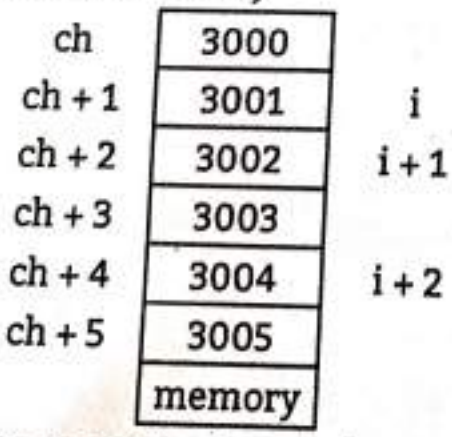
```
int * a;
a = a + a;
```

is also valid.

- Also subtraction of one pointer from another only makes sense when both pointers point to a common object, such as an array. The subtraction then yield number of elements of the base type separating the two pointers values.
- Apart from these operation no other arithmetic operation is valid on pointers. We cannot multiply or divide pointers. We cannot add two pointers.

For example:

```
char *ch = 3000;
int *i = 3001;
```



2. Adding a number to a pointer:

A number can be added to a pointer.

For example:

```
int i = 3, *j;
    j = &i;
    j = j + 1;
    j = j + 9;
```

In line no. 4, j points to 9 integer locations after current location.

5. **Subtracting a number from a pointer:**
A number can be subtracted from a pointer.

For example:

```
int i = 3, *j;
    j = &i;
    j = j - 2;
    j = j - 6;
```

In line no. 4, j points to 6 integer locations before current location.

4. **Subtraction of one pointer from another:**
One pointer variable can be subtracted from another, provided both point to elements of same array. The result is the number of elements between corresponding array elements.

Program 7.4: Program to subtract one pointer from another pointer.

```
#include<stdio.h>
int main( )
{
   int arr [ ] = {10, 20, 30, 45, 67, 56, 74};
   int *i, *j;
   i = &arr[1];
   j = &arr[5];
   printf("%d %d", j - i, * j - * i);
   return 0;
}
```

Output:

```
504/36
```

Here, (j – 1) gives the difference between addresses.
(*j – *i) gives the difference between values contained in addresses at j &i.

5. **Comparison of two pointer variables:**

- Comparison of two pointer variables is possible provided both point to one array (i.e. both pointers point to objects of same data type).
- Pointers can also be compared with zero (NULL).

Program 7.5: Program to compare two pointers variables.

```
#include<stdio.h>
main( )
{
   int arr [ ] = {10, 20, 36, 72, 35, 36};
   int *j, *k;
   j = &arr [4];
   k = (arr + 4);
   if(j = ^ k)
```

```
        printf("\n The two pointers point to same location");
    else
        printf("\n The two pointers do not point to same location");
    return 0;
}
```

Output:

```
The two pointers point to same location.
```

Note: The operations of adding two pointers, multiplication of a pointer with a constant, division of a pointer with a constant should not be performed.

6. Pointer Expressions:

- Pointer variables can be used in expressions like other variables.

For example:

(i) a = (* pt1) * (* pt2);
(ii) s = s + *p1;
(iii) * p2 = * p2 + 15;
(iv) c + = * p3;

Here, we see that ' = ' assignment operator can be used on pointers.

Program 7.6: Write a program for pointer expression.

```
/* Program of pointer expression */
#include<stdio.h>
int main( )
{
    int a = 12; *p;
    p = &a;
    *p = *p + 3;
    printf("%u %d", p, *p);
    getch();
    return 0;
}     /* end of main */
```

Output:

```
6552415
```

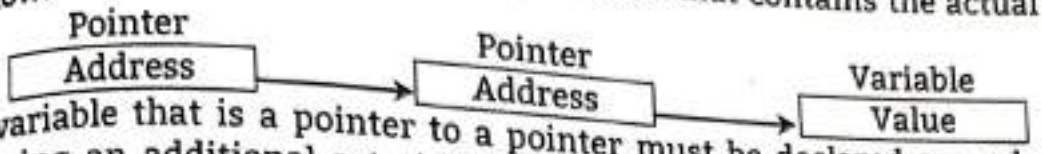
Program 7.7: Program to demonstrate the arithmetical operation pointers.

```
#include<stdio.h>
void main( )
{
    char c='1',*cp;
    float f=10.2,*fp;
    int i=987,*ip;
    long l=345,*lp;
    cp=&c,fp=&f,ip=&lp,lp=&l;
    printf("\n Char float int long");
    printf("\n%8x%8x%8x%8x%8x%",cp,fp,ip,lp");
    for(i=0;i<3;i++)
    {
        cp++;fp++;ip++lp++;
        printf("\n%8x%8x%8x%8x%8x%8x%",cp,fp,ip,lp");
    }
```

```
for(i=0;i<3;i++)
{  cp--;fp--;ip--;lp--;
   printf("\n8x%8x%8x%8x%",cp,fp,ip,lp");
}
}
```

Pointer to Pointer:

- When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below:



- A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

7.4 DYNAMIC MEMORY ALLOCATION [W-14, 15, 16; S-15, 16, 17]

- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Advantages of using Dynamic memory allocation:

1. It prevents overflow and underflow of memory. Sometimes, the memory allocated may be more and sometimes it may be less. DMA allows us to shrink or expand the allocated memory. Hence, there will be no overflow or underflow.
2. Programmer doesn't need to know about required memory space. So it prevents unnecessary updating of program.

Disadvantages of using Dynamic memory allocation:

1. Program often becomes long and complex. Using array is much ore simpler and easier than using functions like malloc, calloc, realloc and free.
2. Memory fragmentation: Sometimes it may be a case that we have sufficient memory as required but they can't be used because they are fragmented.
3. User is responsible for freeing up memory when finished with it.

- Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

1. malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space.
2. calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory.
3. free()	dellocate the previously allocated space.
4. realloc()	Change the size of previously allocated space.

1. **malloc():** The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

   Syntax of malloc(): `ptr=(cast-type*)malloc(byte-size)`

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

For example: `ptr=(int*)malloc(100*sizeof(int));`

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

2. **calloc()**: The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

   Syntax of calloc(): `ptr=(cast-type*)calloc(n,element-size);`

   This statement will allocate contiguous space in memory for an array of n elements.

   For example: `ptr=(float*)calloc(25,sizeof(float));`

   This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

3. **free()**: Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

   Syntax of free(): `free(ptr);`

   This statement cause the space in memory pointer by ptr to be deallocated.

Program 7.8: Program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
if(ptr==NULL)
 {
printf("Error! memory not allocated.");
exit(0);
 }
printf("Enter elements of array: ");
for(i=0;i<n;++i)
 {
scanf("%d",ptr+i);
sum+=*(ptr+i);
 }
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

Output:

```
Enter number of elements: 4
Enter elements of array: 1, 2, 4, 6
Sum = 13
```

Program 7.9: Program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)calloc(n,sizeof(int));
if(ptr==NULL)
 {
printf("Error! memory not allocated.");
exit(0);
 }
printf("Enter elements of array: ");
for(i=0;i<n;++i)
 {
scanf("%d",ptr+i);
sum+=*(ptr+i);
 }
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

Output:

```
Enter number of elements: 6
Enter elements of array: 5 5 6 3 2 4
Sum = 25
```

4. realloc() function:

- If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().
- **Syntax of realloc():** `ptr=realloc(ptr,newsize);`
  Here, ptr is reallocated with size of newsize.

Program 7.10: Program for realloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *ptr,i,n1,n2;
```

```
printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
printf("%u\t",ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=realloc(ptr,n2);
for(i=0;i<n2;++i)
printf("%u\t",ptr+i);
return 0;
}
```

Output:

```
Enter size of array: 2
Address of previously allocated memory: 1894 1896
Enter new size of array: 3
   1894 1896 1898
```

Program 7.11: Program to find largest number using dynamic memory allocation.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
inti,n;
float *data;
printf("Enter total number of elements(1 to 100): ");
scanf("%d",&n);
data=(float*)calloc(n,sizeof(float)); /* Allocates the memory for 'n'
elements */
if(data==NULL)
 {
printf("Error!!! memory not allocated.");
exit(0);
 }
printf("\n");
for(i=0;i<n;++i) /* Stores number entered by user. */
 {
printf("Enter Number %d: ",i+1);
scanf("%f",data+i);
 }
for(i=1;i<n;++i) /* Loop to store largest number at address data */
 {
if(*data<*(data+i)) /* Change <to> if you want to find smallest number */
 *data=*(data+i);
 }
printf("Largest element = %.2f",*data);
return 0;
}
```

Output:

```
Enter total number of elements(1 to 100): 12
Enter Number 1: 2.34
Enter Number 2: 3.43
Enter Number 3: 6.78
Enter Number 4: 2.45
Enter Number 5: 7.64
Enter Number 6: 9.05
Enter Number 7: -3.45
Enter Number 8: -9.99
Enter Number 9: 5.67
Enter Number 10: 34.953
Enter Number 11: 4.5
Enter Number 12: 3.45
Largest element = 34.95
```

7.5 FUNCTIONS AND POINTERS [S-17]

- The general **syntax for declaring a pointer to a function** is,
  `return-type (*pointer-variable) ();`
- The **general syntax of assignment of a pointer to function** is,
  `pointer-variable = function name;`
- This assigns the address of function i.e. address of the first statement of the function to the pointer-variable.
- The function then can be invoked as `(* pointer-variable) ();`

Program 7.12: Program to illustrate pointer to function.

```
#include<stdio.h>
main()
{
   int display ();
   int (*func_ptr) ();
   func_ptr = display;
   /* Assign address of function */
   printf ("\n Address of function display is %d", func-ptr);
   (*func_ptr) ();
   /* invokes function display */
}
int display()
{
   puts ("\n Into the world of pointer to function");
}
```

Output:

```
Address of function display is 679
```

Into the world of pointer to function

- In the above example, func-ptr is a pointer to function which returns an integer. The (*func-ptr) () invokes the function display since func-ptr now points to display.

Program 7.13: Write a program to count the number of words, lines and characters in a text.

[W-15]

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<conio.h>
/*low implies that position of pointer is within a word*/
#define low 1
/*high implies that position of pointer is out of word.*/
#define high 0
void main()
{
   intnob,now,nod,nov,nos,pos=high;
   char *s;
   nob=now=nod=nov=nos=0;
   clrscr();
   printf("Enter any string:");
   gets(s);
   while(*s!='')
   {
   if(*s==' ') /* counting number of blank spaces. */
   {
   pos=high;
   ++nob;
   }
   else if(pos==high) /* counting number of words. */
   {
   pos=low;
   ++now;
   }
   if(isdigit(*s)) /* counting number of digits. */
   ++nod;
   if(isalpha(*s)) /* counting number of vowels */
   switch(*s)
      {
      case 'a':
      case 'e':
```

```
        case 'i':
        case 'o':
        case 'u':
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
        ++nov;
        break;
        }
    /* counting number of special characters */
    if(!isdigit(*s)&&!isalpha(*s))
    ++nos;
    s++;
    }
    printf("\nNumber of words %d",now);
    printf("\nNumber of spaces %d",nob);
    printf("\nNumber of vowels %d",nov);
    printf("\nNumber of digits %d",nod);
    printf("\nNumber of special characters %d",nos);
    getch();
}
```

Output:

```
Enter any string:NIRALI
Number of words 1
Number of spaces 0
Number of vowels 6
Number of digits 9
Number of special character 228
```

Function Pointer

- A pointer which points to the address of a function is known as function pointer.

Declaration

```
function_return_type(*Pointer_name)(function argument list)
```

For example:

```
double (*p2f)(double, char)
```

Here double is a return type of function, p2f is name of the function pointer an
(double, char) is an argument list of this function. Which means the first argument
this function is of double type and the second argument is char type.

Some points about function pointers:

1. Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code. you can declare a function pointer and assign a function to it in a single statement like this:

```
void (*fun_ptr)(int) = &fun;
```

2. Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
3. You can even remove the ampersand from this statement because a function name alone represents the function address.

Example: `void(*fun_ptr)(int) = fun;`

Programs

Program 1: Swap two numbers using pointers.

```
#include <stdio.h>
// function: swap two numbers using pointers
void swap(int *a,int *b)
{
int t;
 t = *a;
 *a = *b;
 *b = t;
}
int main()
{
int num1,num2;
printf("Enter value of num1: ");
scanf("%d",&num1);
printf("Enter value of num2: ");
scanf("%d",&num2);
//print values before swapping
printf("Before Swapping: num1=%d, num2=%d\n",num1,num2);
//call function by passing addresses of num1 and num2
swap(&num1,&num2);
//print values after swapping
printf("After Swapping: num1=%d, num2=%d\n",num1,num2);
return 0;
}
```

Output:

```
Enter value of num1: 10
Enter value of num2: 20
Before Swapping: num1=10, num2=20
After Swapping: num1=20, num2=10
```

Program 2: Change the value of constant integer using pointers.

```
#include <stdio.h>
int main()
{
const int a=10; //declare and assign constant integer
int *p; //declare integer pointer
 p=&a; //assign address into pointer p
printf("Before changing - value of a: %d",a);
//assign value using pointer
 *p=20;
printf("\nAfter changing - value of a: %d",a);
printf("\nWauuuu... value has changed.");
return 0;
}
```

Output:

```
Before changing - value of a: 10
After changing - value of a: 20
Wauuuu... value has changed.
```

Program 3: Print a string using pointer.

```
#include <stdio.h>
int main()
{
   char str[100];
   char *ptr;
   printf("Enter a string: ");
   gets(str);
   //assign address of str to ptr
   ptr=str;
   printf("Entered string is: ");
   while(*ptr!='\0')
   printf("%c",*ptr++);
   return 0;
}
```

Output:

```
Enter a string : adcbg
Entered string is : adcbg
```

Summary

- A pointer is nothing but a memory location where data is stored.
- A pointer is used to access the memory location.
- There are various types of pointers such as a null pointer, wild pointer, void pointer and other types of pointers.
- Pointers can be used with array and string to access elements more efficiently.
- We can create function pointers to invoke a function dynamically.
- Arithmetic operations can be done on a pointer which is known as pointer arithmetic.
- Pointers can also point to function which make it easy to call different functions in the case of defining an array of pointers.
- When you want to deal different variable data type, you can use a typecast void pointer.

Check Your Understanding

1. A pointer is _____.
   (a) A variable that stores address of an instruction
   (b) A variable that stores address of other variable
   (c) A keyword used to create variables
   (d) None of these
2. The reason for using pointers in a C program is _____.
   (a) Pointers allow different functions to share and modify their local variables.
   (b) To pass large structures so that complete copy of the structure can be avoided.
   (c) Pointers enable complex "linked" data structures like linked lists and binary trees.
   (d) All of the above
3. Address stored in the pointer variable is of type _____.
   (a) Integer (b) Float
   (c) Array (d) Character
4. In order to fetch the address of the variable we write preceding _____ sign before variable name.
   (a) Percent(%) (b) Comma(,)
   (c) Ampersand(&) (d) Asteric(*)
5. Comment on this const int *ptr;
   (a) You cannot change the value pointed by ptr
   (b) You cannot change the pointer ptr itself
   (c) Both (a) and (b)
   (d) You can change the pointer as well as the value pointed by it

Answers

1. (b)	2. (d)	3. (a)	4. (c)	5. (a)

(a)
```
int main()
{
    int a[] = { 1, 2, 3, 4, 5} ;
    int *ptr;
    ptr = a;
    printf(" %d ", *( ptr + 1) );
    return 0;
}
```

(b)
```
int main()
{
    int a = 5;
    int *ptr ;
    ptr = &a;
    *ptr = *ptr * 3;
    printf("%d", a);
    return 0;
}
```

(c)
```
int main()
{
    int i = 6, *j, k;
    j = &i;
    printf("%d\n", i * *j * i + *j);
    return 0;
}
```

(d)
```
int main()
{
    int x = 20, *y, *z;
    // Assume address of x is 500 and
    // integer is 4 byte size
    y = &x;
    z = y;
    *y++;
    *z++;
    x++;
    printf("x = %d, y = %d, z = %d \n", x, y, z);
    return 0;
}
```

(e)
```
int main()
{
    Int x = 10;
    Int *y, **z;
    y = &x;
    z = &y;
    printf("x = %d, y = %d, z = %d\n", x, *y, **z);
    return 0;
}
```

Practice Questions

Q.1 Write the answer of following questions in brief:

1. Enlist various applications of pointers.
2. What are the differences between malloc() and calloc()?
3. Difference between const char* p and char const* p
4. What is a null pointer?
5. How are pointer variables initialized?

Q.2 Write the answer of following questions:

1. What is meant by pointer? State advantages and disadvantages of pointer.
2. Write a program to reverse the contents of an integer/character array using pointers.
3. Write a program to store a matrix of size m × n using dynamic memory allocation and print the transpose of the matrix.
4. What are pointer variables? How are they different than type of variables? Explain with example.
5. Explain the pointer arithmetic with example.

Q.3 Define terms:

1. Pointer, 2. Dynamic memory allocation, 3. Function pointer, 4. malloc(), 5. calloc()

Previous Exam Questions

Winter 2014

1. What is pointer? Explain how to declare and initialize pointer variable with suitable example. [5 M]

Ans. Refer to Section 7.1.

2. What is dynamic memory allocation? Explain its advantages. [5 M]

Ans. Refer to Section 7.4.

Summer 2015

1. What is Indirection operator. [2 M]

Ans. Refer to Section 7.2.

2. What is Dynamic Memory Allocation? Explain functions used to allocate and delete memory dynamically. [5 M]

Ans. Refer to Section 7.4.

Winter 2015

1. Define Pointer? Explain with example. [2 M]

Ans. Refer to Section 7.1.

2. What is use of calloc function? [2 M]

Ans. Refer to Section 7.4.

3. What is Dynamic memory allocation? Explain functions used to allocate and delete memory? [5 M]

4. [5 M]

```
main()
{
    char *p = "a | qc";
    printf("%c...", ++ * (++p));
    printf("%c...", ++ p);
}
```

Output: m q

Summer 2016

1. What is use of malloc() function? [2 M]

Ans. Refer to Section 7.4.

2. What is dynamic memory allocation? Explain functions used to allocate and delete memory dynamically. [5 M]

Ans. Refer to Section 7.4.

3. [5 M]

```
main( )
{
    char * m = "ABCD";
    printf("%C---", ++ * (++ p));
    printf("%C", * ++ p);
}
```

Winter 2016

1. What is Indirection Operator? [2 M]

Ans. Refer to Section 7.2.

2. Write a 'C' program for multiplication of two matrices using dynamic memory allocation. [5 M]

Ans. Refer to Program 5.

Summer 2017

1. Dynamic memory allocation. Explain various functions used for the same with their syntax. [5 M]

Ans. Refer to Section 7.4. [5 M]

2. Function pointer.

Ans. Refer to Section 7.5.

Summer 2018

1. Is the difference between malloc() and calloc()? [2 M]

Ans. Refer to Section 7.4.

2. What is generic pointer in C? [2 M]

Ans. When a variable is declared as being a pointer to type void it is known as a generic pointer. A generic pointer cannot be directly dereferenced. We need to typecast it to relevant data type before dereferencing.

3. Trace output [5 M]

```
#include<stdio.h>
main()
{
   int arr[ ] = {0, 1, 2, 3, 4};
   int i, * ptr;
   for(ptr = arr +4) i = 0; i ≤ 4; i++)
   printf("%d", ptr [- - i]);
}
```

8... Structures

Learning Objectives...

- To study the definition and declaration of structure.
- To study, How to access the different variables via a single pointer?
- To learn array of structure.

8.1 INTRODUCTION TO STRUCTURE [S-16]

- Structure is analogous to records. A structure is a user defined data type. A structure is a convenient tool for handling a group of logically related data items.
- Structure is a collection of logically related data items of different data types grouped together under a single name.

8.1.1 Definition [W-15, S-15, 18]

- Structure is user defined data type which is used to store heterogeneous data under unique name.

OR

- Structure is a collection of different data types which are grouped together and each element in a structure is called member.
- To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.
- The format of the struct statement:

```
struct [structure tag]
{
 member definition;
 member definition;
 ...
 member definition;
} [one or more structure variables];
```

- The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more

structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
 char title[50];
 char author[50];
 char subject[100];
 int book_id;
} book;
```

8.1.2 Declaration and Initializing Structure [W-16, S-16]

1. Declaration of Structure:

- To define or declare a structure you can use struct keyword.

Syntax:

```
struct structure_name
{
 <data-type> element 1;
 <data-type> element 2;
    :
 :
 <data-type> element n;
}struct_var;
```

Example:

```
struct emp_info
{
 char emp_id[10];
 char name[100];
 float sal;
}emp;
```

Note:

- Structure is always terminated with semicolon (;).
- Structure name as emp_info can be later used to declare structure variables of its type in a program.

2. Initializing Structure:

- C programming language treats a structure as a custom data type therefore you can initialize a structure like a variable.
- Here, is an example of initialize product structure:

```
struct product
{
  char name[50];
  double price;
} book = {"C programming language", 40.5};
```

- In above example, we define product structure, then we declare and initialize.

How to Declare Structure Variables?

- We are already seen how to declare structure. But we cannot use it in the program, since it is not a variable name. Hence, we cannot assign any values, cannot perform arithmetic and logical operations.
- To perform all such actions, we need to associate structure variable with the structure type.
- There are three methods of associating structure variables with the structure-type.
  (a) Structure-variable names in structure declaration.
  (b) Structure-variables as data declarations.
  (c) Array of structure-variables.

1. Structure-variable Names in Structure Declaration:

- The structure declaration may contain structure variable names for structure-variables.

Syntax:

```
struct structure-name
{
data-type1     element 1;
data-type2     element 2;
:
:
data-type n    element n;
} var 1, var 2, ..... var n;
```

For example:

```
struct employee
{
   char empname[20];
   char empcode[6];
   float basic;
}  emp1, emp2, emp3;
```

Here, emp 1, emp 2, emp 3 are three variables of structure type employee.

2. Structure-variables as Data Declarations:

- A structure variables declaration can be made as data type declaration since structure-type is a programmer derived data type.

For example:

```
struct employee
{
   char empname[20];
   char empcode[6];
   float basic;
};
main ()
static struct emp1, emp2, emp3;
```

Where, emp1, emp2, emp3 are three variables of structure type-employee.

3. **Array of Structure Variables:**
   - If we try to use the methods above for declaring 100 employee's details, we will require 100 structure variable names which is not possible.
   - The solution is that to declare arrays. Structure variable declaration can be made as array of structure-variable declaration.

   For example:

```
struct employee
{
    char empname[20];
    char empcode[6];
    float basic;
};
struct employee emp[100];
```

8.1.3 Accessing Members of Structures

- The variables which are declared inside the structure are called as members of structure.
- Structure members can be accessed using member operator '.' . It is also called as 'dot operator' or 'period operator'.

Syntax:

```
structure_var.member;
```

Program 8.1: Program to demonstrate structure.

```
#include<stdio.h>
#include<conio.h>
struct comp_info
{
    char nm[100];
    char addr[100];
}info;
void main()
{
    clrscr();
    printf("\n Enter Company Name: ");
    gets(info.nm);
    printf("\n Enter Address: ");
    gets(info.addr);
    printf("\n\n Company Name: %s",info.nm);
    printf("\n\n Address: %s",info.addr);
    getch();
}
```

Output:

```
Enter Company Name: Nirali Prakashan
Enter Address: Pune, Maharashtra, INDIA
Company Name: Nirali Prakashan
Address: Pune, Maharashtra, INDIA
```

Program 8.2: Program for structure.

```
#include <stdio.h>
#include <string.h>
struct Books
{
 char title[50];
 char author[50];
 char subject[100];
 int book_id;
};
int main( )
{
 struct Books Book1; /* Declare Book1 of type Book */
 struct Books Book2; /* Declare Book2 of type Book */
 /* book 1 specification */
 strcpy( Book1.title, "C Programming");
 strcpy( Book1.author, "Kamil Khan");
 strcpy( Book1.subject, "C Programming Tutorial");
 Book1.book_id = 1121;
 /* book 2 specification */
 strcpy( Book2.title, "Basic of C++");
 strcpy( Book2.author, "Sallaudin Sajjan");
 strcpy( Book2.subject, "Basics of C++ Tutorial");
 Book2.book_id = 2172;
 /* print Book1 info */
 printf( "Book1 title: %s\n", Book1.title);
 printf( "Book1 author: %s\n", Book1.author);
 printf( "Book1 subject: %s\n", Book1.subject);
 printf( "Book1 book_id: %d\n", Book1.book_id);
 /* print Book2 info */
 printf( "Book2 title: %s\n", Book2.title);
 printf( "Book2 author: %s\n", Book2.author);
 printf( "Book2 subject: %s\n", Book2.subject);
 printf( "Book2 book_id: %d\n", Book2.book_id);
getch();
return 0;
}
```

Output:

```
Book 1 title: C Programming
Book 1 author: Kamil Khan
Book 1 subject: C Programming Tutorial
Book 1 book_id: 1121
Book 2 title: Basic of C++
Book 2 author: Sallaudin Sajjan
Book 2 subject: Basics of C++ Tutorial
Book 2 book_id: 2172
```

Program 8.3: Program to demonstrate the initialization of structures.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct student
    {
      int roll_no;
      char name[25];
      float per;
    };
struct student s={10,"Kamil",84.27};
printf("\n Roll No=%d",s.roll_no);
printf("\n Name =%s",s.name);
printf("\n Percentage =%f",s.per);
getch();
}
```

Output:

```
Roll No = 10
Name = Kamil
Percentage = 84.269997
```

Program 8.4: Program to accept structure members from the user and display them.

```
#include<stdio.h>
void main()
{
   struct book
   {
      int book_id;
      char book_name[10];
      float  price;
   };
printf("\n Enter book-id, name and price: \n");
scanf("%d%s%d", &s.book-id, s.book_name, &s.price);
printf("\n Entered information: ");
printf("\n Book_ID: %d", s.book_id);
printf("\n Book_Name: %s", s.book_name);
printf("\n Price: %d",s.price);
getch();
}
```

Output:

```
Enter book-id, name and price:
1265   C 225
Entered information:
Book_ID: 1265
Book_Name: C
Price: 225
```

Program 8.5: Program to Declare a structure 'struct_time' having data members hour, minutes, seconds. Accept this data and display the time in format 18:25:30.

```
#include<stdio.h>
#include<conio.h>
struct struct_time
{
   int hour;
   int minutes;.
   int seconds;
}T;
void main()
{
clrscr();
printf("\n Enter Hours : ");
scanf("%d",&.hour);
printf("\n Enter Minutes : ");
scanf("%d",&T.minutes);
printf("\n Enter Seconds : ");
scanf("%d",&T.seconds);
printf("\n Time is : %d:%d:%d",T.hour,T.minutes,T.seconds);
getch();
}
```

Output:

```
Enter Hours : 11
Enter Minutes : 32
Enter Seconds : 45
Time is : 11:32:45
```

Program 8.6: Program to Declare a structure describing 'circle' having data members as radius, perimeter and area. Accept radius and display area and perimeter.

```
#include<stdio.h>
#include<conio.h>
struct circle
{
int radius;
float peri;
float area;
}a;
void main()
{
clrscr();
printf("\n Enter radius : ");
scanf("%d",&a.radius);
a.peri = 2.0 * 3.14 * (float) a.radius;
printf("\n Perimeter : %.2f",a.peri);
a.area = 3.14 * (float)(a.radius * a.radius);
printf("\n Area: %.2f",a.area);
getch();
}
```

Output:

```
Enter radius : 12
Perimeter : 75.36
Area: 452.16
```

Array of Structures:

- Structure are often arrayed.
- To declare an array of structures, you must first define a structure and then declare an array variable of that type.
- General **syntax** of array,

```
data-type array-name [size of array];
```

For example:

```
struct emp elist [50];
```

- Here we declare array of structure for employee. We access array elements using index of array i.e. num[0], num[1] ..... etc. But here elements of array are structures and each number of structure is accessed by '.' operator. Hence, member can be obtained using,

```
array-name [index].member-name
```

- Hence, to access basic of 3rd employee, we say elist[2].basic (if starting index is '0'). elist[2] gives the entire structure for 3rd employee & elist[2] basic gives salary for 3rd employee.

Program 8.7: Program to calculate net pay of employees.

```
/* A program to calculate net pay */
#include<stdio.h>
void main()
{
struct emp
{
 char ename[20];
 char ecode[6];
 float basic;
 float hra, da;
 float npay;
} e[50];
 int i;
 for(i=0;i<1;i++)
 {
printf("\n Enter name of employee:");
scanf("%s", &e[i].ename);
printf("\n Enter ecode of employee:");
scanf("%s", &e[i].ecode);
printf("\n Enter basic pay:");
scanf("%f", &e[i].basic);
```

```
printf("\n Enter HRA:");
scanf("%f", &e[i].hra);
printf("\n Enter DA:");
scanf("%f", &e[i].da);
}
for(i=0;i<1;i++)
{
e[i].npay = e[i].basic + e[i].hra + e[i].da;
printf("\n Name Ecode Basic pay HRA DA Net pay");
printf("\n %s %s %f %f %f %f",
e[i].ename, e[i].ecode, e[i].basic, e[i].hra, e[i].da, e[i].npay);
}
}
```

Output:

```
Enter name of employee : Pradeep
Enter code of employee : 101
Enter basic pay : 8000
Enter HRA : 100
Enter DA : 200
Name      Ecode     Basic Pay     HRA       DA        Net Pay
Pradeep   101       8000.00       100.0     200.0     8300.00
```

8.2 OPERATIONS ON STRUCTURES

- There is a relatively small number of operations which C directly supports on structures. As we have seen, we can define structures, declare variables of structure type and select the members of structures.
- We can also assign entire structures: the expression,

```
c1 = c2
```

would assign all of c2 to c1 (both the real and imaginary parts, assuming the preceding declarations). We can also pass structures as arguments to functions and declare and define functions which return structures. But to do anything else, we typically have to write our own code (often as functions). For example, we could write a function to add two complex numbers:

```
struct complex
cpx_add(struct complex c1, struct complex c2)
{
  struct complex sum;
  sum.real = c1.real + c2.real;
  sum.imag = c1.imag + c2.imag;
  return sum;
}
```

We could then say things like,

```
c1 = cpx_add(c2, c3)
```

One more thing you can do with a structure is initialize a structure variable declaring it. As for array initializations, the initializer consists of a comma-separated list of values enclosed in brances { }:

```
struct complex c1 = {1, 2};
struct complex c2 = {3, 4};
```

- The type of each initializer in the list must be compatible with the type of the corresponding structure member.

8.3 NESTED STRUCTURE [S-15, 18]

- C language allows a member of a structure can be a structure itself. Such structure declarations with some members as themselves being structures is called as **nested/ embedded structure**.
- The nested structure must be declared before it is declared as a member of another structure.
- When a member of a structure has to be further broken down into entities, that member can be declared as structure.
- Nested structure are used in such events, where multiple data items related to each other, are repeated within a structure.

Syntax:

```
struct structure_name
{
   <data-type> element 1;
   <data-type> element 2;
   - - - - - - - - - - -
   - - - - - - - - - - -
   <data-type> element n;
struct structure_name
   {
      <data-type> element 1;
      <data-type> element 2;
      - - - - - - - - - - -
      - - - - - - - - - - -
      <data-type> element n;
   } inner_struct_var;
} outer_struct_var;
```

Program 8.8: Program to demonstrate nested structures.

```
#include<stdio.h>
#include<conio.h>
struct stud_Res
{
   int rno;
   char std[10];
```

```
    struct stud_Marks
    {
        char subj_nm[30];
        int subj_mark;
    }marks;
}result;
void main()
{
    clrscr();
    printf("\n\t Enter Roll Number: ");
    scanf("%d",&result.rno);
    printf("\n\t Enter Standard: ");
    scanf("%s",result.std);
    printf("\n\t Enter Subject Code: ");
    scanf("%s",result.marks.subj_nm);
    printf("\n\t Enter Marks: ");
    scanf("%d",&result.marks.subj_mark);
    printf("\n\n\t Roll Number: %d",result.rno);
    printf("\n\n\t Standard: %s",result.std);
    printf("\n\n\tSubject Code: %s",result.marks.subj_nm);
    printf("\n\n\t Marks: %d",result.marks.subj_mark);
    getch();
}
```

Output:

```
Enter Roll Number: 1
Enter Standard: BCAII
Enter Subject Code: SUB001
Enter Marks: 63
Roll Number: 1
Standard: BCAII
Subject Code: SUB001
Marks: 63
```

Program 8.9: Write a C program to add two distances entered by user. Measurement o distance should be in inch and feet.(Note: 12 inches = 1 foot).

```
#include <stdio.h>
struct Distance
{
int feet;
float inch;
}d1,d2,sum;
```

```
int main()
{
printf("1st distance\n");
printf("Enter feet: ");
scanf("%d",&d1.feet); /* input of feet for structure variable d1 */
printf("Enter inch: ");
scanf("%f",&d1.inch); /* input of inch for structure variable d1 */
printf("2nd distance\n");
printf("Enter feet: ");
scanf("%d",&d2.feet); /* input of feet for structure variable d2 */
printf("Enter inch: ");
scanf("%f",&d2.inch); /* input of inch for structure variable d2 */
sum.feet=d1.feet+d2.feet;
sum.inch=d1.inch+d2.inch;
if (sum.inch>12){ //If inch is greater than 12, changing it to feet.
++sum.feet;
sum.inch=sum.inch-12;
}
printf("Sum of distances=%d\'-%.1f\"",sum.feet,sum.inch);
/* printing sum of distance d1 and d2 */
return 0;
getch();
}
```

Output:

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distance = 15' - 5.1"
```

Program 8.10: Program for calculating area of the circle using structure. [W-14 to S-17]

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct circle
   {
      float area;
      float radius;
   };
```

```
    struct circle one;
    printf("\n Enter radius:");
    scanf("%f",&one.radius);
    one.area=3.1415*one.radius*one.radius;
    printf("\n area = %f ",one.area);
    getch();
}
```

Output:

```
Enter radius: 12
area = 452.376007
```

Summary

- A structure is usually used when we wish to store dissimilar data together.
- Structure elements can be accessed through a structure variable using a dot (.) operator.
- Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
- All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
- It is possible to pass a structure variable to a function either by value or by address.
- It is possible to create an array of structures.

Check Your Understanding

1. Which of the following are themselves a collection of different data types?
   (a) String (b) Structures
   (c) Char (d) None of the above
2. Which operator connects the structure name to its member name?
   (a) - (b) ->
   (c) . (d) both . and ->
3. Which of the following cannot be a structure member?
   (a) Function (b) Array
   (c) Structure (d) None of the above
4. What is the correct syntax to declare a function foo() which receives an array of structure in function?
   (a) void foo(struct *var); (b) void foo(struct *var[]);
   (c) void foo(struct var); (d) none of the mentioned
5. Which of the following accesses a variable in structure b?
   (a) b->var; (b) b.var;
   (c) b-var; (d) b>var;

Answers

1. (b)	2. (c)	3. (a)	4. (a)	5. (b)

Trace the Output

(a)
```
struct sample
{
    int a = 0;
    char b = 'A';
    float c = 10.5;
};
int main()
{
    struct sample s;
    printf("%d, %c, %f", s.a, s.b, s.c);
    return 0;
}
```

(b)
```
int main()
{
    struct bitfield
    {
        signed int a : 3;
        unsigned int b : 13;
        unsigned int c : 1;
    };
    struct bitfield bit1 = { 2, 14, 1 };
    printf("%d", sizeof(bit1));
    return 0;
}
```

(c)
```
int main()
{
    typedef struct tag
    {
        char str[10];
        int a;
    } har;
    har h1, h2 = { "IHelp", 10 };
    h1 = h2;
    h1.str[1] = 'h';
    printf("%s, %d", h1.str, h1.a);
    return 0;
}
```

(d)
```
struct sample
{
    int a;
} sample;
int main()
{
    sample.a = 100;
    printf("%d", sample.a);
    return 0;
}
```

Practice Questions

Q.1 Write the answer of following questions in brief.

1. What are the differences between structures and arrays?
2. Rules for declaring a structure?
3. Define structure pointers.
4. How to declare structure.
5. How to access member of structure.

Q.2 Write the answer of following questions

1. With suitable example explain the nested structure.
2. What is the use of a structure? Explain with example.
3. How to declare and initialise structure? Give an example.
4. Can a function return a value of type 'pointer to structure'?
5. Write a program to read roll numbers and names and total marks of 10 students. Display the data alphabetically sort on name.
6. A structure stores the details of 15 books. Details include the title, price, number of pages, publication and popularity in grades.

   Grade 'A' for highest popularity
   'B' for average
   'C' for low popularity.

7. Write a program to print details of the books which has highest popularity.

Q.3 Define terms:

(a) Structure
(b) Nested structure
(c) Array of structure variable.

Previous Exam Questions

Winter 2014

1. What is nested structure? [2 M]

Ans. Refer to Section 8.3.

2. Create a structure to store data of 10 students as roll no, name and percentage. Write a 'C' program to print roll no, names of students who have succeed less than 60 percent. [5 M]

Ans. Refer to Program 8.8.

Summer 2015

1. Give syntax to define self Referential structure. [2 M]

Ans. A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. [2 M]

2. Define structure with an example.

Ans. Refer to Section 8.1.1.

3. Create structure to store data of 10 employees as employee number, name and salary. Write a 'C' program to print Employee Numbers and names of Employees having salary greater then 10,000. [5 M]

Ans. Refer to Program 8.7.

Winter 2015

1. What is structure? Explain with example. [5 M]

Ans. Refer to Section 8.1.

2. Create a structure to store detail of 10 students as Roll No, name and percentage. Write a 'C' program to print Roll No, Names of students who have secured more than 70 percent. [5 M]

Ans. Refer to Program 8.8.

Summer 2016

1. Define structure. Give suitable example. [2 M]

Ans. Refer to Sections 8.1.1 and 8.1.2.

2. Write a 'C' program to accept and display book details of 'n' books as book-title, author, publisher and cost. (using array of structure). [5 M]

Ans. Refer to Program 8.2.

3. [5 M]

```
main()
{
   struct student
   {
   char name [20];
   int rollno;
   }
   S1, * ptr, S[10];
   printf("\n %d", size of (S1));
   printf("\n %d", size of (ptr));
   printf("\n %d", size of (S));
}
```

Winter 2016

1. What is array of structure? Give example. [2 M]

Ans. Refer to Section 8.1.3.

Summer 2017

1. Write a program to accept name, author, rate, quantity of n books from user and display name of book author name and total cost. [5 M]

Ans. Refer to Program 8.2.

Summer 2018

1. What is structure? [2 M]

Ans. Refer to Section 8.1.1.

2. Define structure. What do you mean by Nested structure? Explain with example. [5 M]

Ans. Refer to Sections 8.1 and 8.3.

■■■