UNIT 1

Syllabus

Introduction to 'C' Language: History, structures of 'C' programming, function as building blocks.

Language Fundamentals: Character set, C tokens, keywords, indentifiers, variables, constants, data types, comments.

Section-A *(Very Short Answer Questions)*

Q.1. Give any four features of 'C'.

Or **What are the general characteristics of C?** *(2017)*

Ans. C is one of the most popular computer languages that is reliable and easy to use. It has the following four features:

(a) General purpose programming language.

(b) Structured programming language.

(c) Limited number of keywords.

(d) Reduces the gap between high-level and low-level langauge.

Q.2. Why C language is known as procedural language? *(2014, 16)*

Ans. Procedural programming uses a list of instructions to tell the computer what to do step-by-step. Procedural programming relies on-you guessed it-procedures, also known as routines or subroutines. A procedure contains a series of computational steps to be carried out. Procedural programming is also referred to as imperative programming. Procedural programming languages are also known as top-down languages.

Procedural programming is intuitive in the sense that it is very similar to how you would expect a program to work. If we want a computer to do something, we should provide step-by-step instructions on how to do it. It is, therefore, no surprise that most of the early programming languages are all procedural. Examples of procedural languages include Fortran, COBOL and C.

Q.3. Write a brief note on the character set used in the 'C' language.

Ans. In most basic sense, a 'C' program is a sequence of characters, which belong to the American Standard Code for Information Interc}hange (ASCII) set. The character set can be divided into following groups:

(a) Alphabets : The 26 uppercase and 26 lowercase letters of the alphabets, *i.e.*, A–Z and a–z.

(b) Digits: The 10 numerals (0–9).

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

(d) White Space Characters: Space, tab, carriage return, formfeed and newline.

However, the compiler ignores the white space characters.

Q.4. Write a brief note on the tokens used in the 'C' language.

Or **What do you understand by C tokens?** *(2016)*

Ans. In general text, a word and punctuation marks are called tokens but in C language, the smallest individual units are called tokens. They are made up of alphabets, digits, white space characters and special characters used in the 'C' language. Some of the widely used tokens are —constants, keywords, identifiers and operators. These tokens are combined to form instructions for the 'C' programs. A group of instructions would be combined later on to form a program.

Q.5. Define keywords. Name any five keywords of C language. *(2013)*

Ans. Keywords are the identifiers whose meaning is predefined to the system. They are reserved for doing specific tasks. They are also identifiers but cannot be user defined since they are reserved words. Five keywords of C language are—`int`, `float`, `double`, `short` and `struct`.

Q.6. Write the syntax of redefining data types using typedef keyword.

Ans. Syntax:
```
typedef type-definition identifier;
```

Q.7. "Extern" keyword is used for _____ ? *(2014)*

Ans. The '`extern`' keyword is used primarily for variable declarations. When we forward declare a function, the keyword is optional.

Q.8. What are variables?

Ans. A quantity which may vary during program execution is called a variable. Variable names are the names that are given to location in the memory of computer where different literals (or constant values) are stored.

Q.9. What are signed qualifiers?

Ans. By default, every variable is allotted a sign bit. If a variable is type char, then left most bit is used as sign bit and remaining 7 bits are used as data bits. If the sign bit is 1, the number is negative and if it is 0, the number is positive. Such variables are called signed qualifiers.

Q.10. What is `const` modifier?

Ans. The constant is a keyword, which makes variable values unmodifiable. The `const` modifier enables us to assign an initial value to a variable that cannot later be changed by the program.

Syntax:
```
const  datatype variable-name (= value);
```

Q.11. What is exponential form?

Ans. In exponential form, a real constant has two parts—a mantissa and an exponent. The mantissa must be either an integer or a real constant, followed by a letter E or e. The exponent must be an integer constant, *e.g.* 152 E + 03.

Q.12. What are hexadecimal (base 16) integer literals?

Ans. Hexadecimal integer literals consist of any combination of digits from the set 0 through a and A to F (or a through f) with a leading Ox or OX. Here letters 'a' through 'f' substitutes the number 10 through 15.

Q.13. Define data types.

Ans. Data types are a finite set of values along with set of rules for permissible operations. Thus, a data type is just an interpretation applied to a string of bits.

Q.14. Write down the four categories of data types.

Ans. Following are the four categories of data types:

(a) Simple data types (Integers, real and character).

This statement tells the compiler "A function that looks like this is coming up later in the program so its all right if you see references to it before you see the function definition itself".

We can eliminate the function declaration if the function definition (the function itself) appears in the listing before the first call to the function. A function can be passed on some values to work on. These values are called arguments (or parameters). When referencing (or invoking) a function, it can be followed by one or more arguments enclosed in parentheses and separated by commas. These arguments are called actual arguments. The actual arguments can be constants, variable names, subscripted variables, or expressions.

The actual arguments will transfer information to formal arguments within the function definition. These arguments here are called formal arguments or dummy arguments. The formal arguments must be variables. The actual arguments, if any, must correspond in number, type and order with formal arguments.

The return statement in a called function returns the value to the calling function. While many arguments may be sent to a function, only one argument may be returned from it.

Q.2. What are the different data types in C? Explain each with an example. ***(2011)***

Ans. Different Data Types in 'C': C has a concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location. The value of a variable can be changed any time.

The different data types in C are:

S.No.	Data type	Uses	Syntax
1.	`int`	`int` is used to define integer numbers.	`{ int count; count = 5; }`
2.	`float`	`float` is used to define floating point numbers.	`{ float miles; miles=7.8; }`
3.	`double`	`double` is used to define big floating point numbers. It reserves twice the storage for the number. On PCs, this is likely to be 8 bytes	`{ double satoms; atoms=250000000; }`
4.	`char`	`char` defines characters	`{ char n; n = 'a'; }`

Q.3. Why C language is known as rich and powerful in its data types? Give examples of each. ***(2014)***

Ans. C language is rich in data types. ANSI C supports three classes of data types:

1. Primary data type (fundamental).
2. Derived data types.
3. User defined data types.

All 'C' compilers support five fundamental data types:

1. Integer (`int`),
2. Character (`char`),
3. Floating point (`float`),
4. Double-precesion (`double`),
5. `void`.

Different Compilers of 'C' Language

Different organisations have written and implemented compilers for 'C' language. Even for the same computer, there may be several compilers, each with their own specific requirements. For example, for IBM PC (International Business Machine – Personal Computer), we have Microsoft C compiler and Quick C compiler, both developed by Microsoft Corporation and Turbo C compiler from Borland International. Similarly for machine running UNIX operating system, we have another compiler named ANSI C. Some other compilers are—Lattice C, Aztec C and Lightspeed C. There may be only a little difference between different compilers of 'C'.

Q.2. Explain the format of 'C' program.

Or **Explain the block structure of a 'C' program.** *(2012)*

Ans.

General Structure (or Format) of a 'C' Program

```
Pre-processor directives, if any
Global variable declarations, if any
Function defining statement
{       Local variables
        Function declaration
        Executable statements
}
```

Thus, the general structure of a 'C' program comprises of the following:

1. Functions: 'A function is a self-contained program structure that performs a specific task'.

Each function has a declarator or function defining statement, *i.e.* name, an optional list of input parameters (also called arguments of the function) with their data type, (a return data type) and a compound statement (also known as function body).

Any 'C' program contains at least one function. If a program contains only one function, it must be `main()`. If there are more than one function, then one of these must be `main()`, because program execution always begins with `main()`. There is no limit on the number of functions present in a 'C' program. There are three things related with a 'C' function—its definition, its declaration and its calling.

2. Function Declaration: It tells the compiler "A function that looks like this is coming up later in the program so its all right if you see references to it before you see the function definition itself".

We can eliminate the function declaration if the function definition (the function itself) appears in the listing before the first call to the function.

3. Statements: A 'C'-statement is a valid 'C'-expression delimited by semi-colon. A compound statement is a group of zero or more statements enclosed within curly braces {}. Compound statements may be nested, *i.e.* one compound statement may exist inside another one. Each instruction in a 'C' program is written as a separate statement. Therefore, a complete 'C' program will comprise of a series of statements.

Some rules of statements are as follows:

(a) Blank spaces may be inserted between two words to improve the readability of the statement.

(b) 'C' is a case-sensitive programming language, *i.e.* it will treat INT, Int and int differently.

(c) 'C' is a free-form programming language. So, a 'C' program is written in free format. The 'C' compiler does not care about where we begin typing on the line. A statement can span several lines and the same line can contain several statements.

(d) Mostly 'C' statements always end with a semi-colon (;).

(e) A set of statements enclosed in braces is called a compound statement.

and is one of the several [illegible]

Comment about the program is enclosed within /* and */. Between these pair of characters (also called delimiters) any characters may be included in either uppercase or lowercase.

Q.3. Write short notes on the following:

I. Comments, ***(2013)***

II. Keywords,

III. Constants,

IV. Names/Identifiers.

Or **What is an identifier? Explain the rules to coin identifier name.** ***(2011)***

Or **What are keywords in C? Explain their importance.** ***(2012)***

Or **What do you understand by identifiers?** ***(2015)***

Or **Define the following terms with suitable example:** ***(2017)***

(a) Constant.

(b) Identifier.

Ans.

I. Comments

They have the following features:

1. Comments about the program should be enclosed within /* and */. Between these pair of characters (also called delimiters), any characters may be included in either uppercase or lowercase.
2. Comments can also be given with //.
   Example: `// Program 1`
   All the text given after the // will be treated as the comment.
3. Comments cannot be nested. For example, `/*Calculation /*sum*/ */` is invalid.
4. Any number of comments can be given at any place in the program.
5. The comment is a non-executable statement.
6. A comment can be split over more than one line.

II. Keywords

Keywords, also called reserved words, are the words whose meaning has already been explained to the 'C' compiler. There are only 32 keywords available in 'C'. All the keywords are in lower case. The keywords cannot be used as identifier names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the 'C' compiler. The list of keywords available in 'C' is:

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

Importance of Keywords: Keywords are strictly explicit and reserved identifiers which have a definite meaning that cannot be changed by any user. Some of the keyboards are compiler specific regarding implementation. Each keyword in a 'C' program is used to create a meaning or convey something that is of high importance. They also increase efficiency and accessibility of a 'C' program.

III. Constants

Constants (also known as literals) in 'C' are a sequence of characters that represent constant values that do not change. This quantity can be stored at a location in the memory of computer. Four major types of constants are—Integer, Floating-point, Character and String.

1. **Integer Constants:** An integer constant refers to a sequence of one or more digits. It must not contain decimal point. It may contain either + or – sign. A number with no sign is assumed to be positive. Spaces, commas and non-digit characters are not permitted between digits.
   'C' allow us to write integer constants in three ways:
   (a) Decimal (Base 10) Integer Literal : It consists of any combination of digits from the set 0 through 9. For example,

```
345  0  -49  +5
```

(b) Octal (Base 8) Integer Literal: It consists of any combination of digits from the set 0 through 7 with a leading 0. For example,

```
034   034    071   00
```

(c) Hexadecimal (Base 16) Integer Literals: It consists of any combination of digits from the set 0 through 9 and A to F (or a through f) with a leading 0x or 0X. Here, letters A through F substitutes the numbers 10 through 15. For example,

```
0x45   0x3   0XA9   0x0
```

Some invalid integer constants are as follows:

`32,000`	(Comma is not allowed)
`12 890`	(Space is not allowed)
`0495`	(9 is not allowed in octal integer literal)
`0x 3A`	(Space is not allowed)
`0x3AH`	(H is not allowed in hexadecimal integer literal)

2. Floating-point (or Real) Constants: A real constant refers to a sequence of one or more digits. It must contain a decimal point. This constant is used to represent values that have fractional part. It can be written in the following two ways:

(a) Fractional Form : In this form, a real constant must have one digit before the decimal point and one digit after the decimal point. It may also have + or – sign preceding it. Without the sign, a real constant is assumed to be positive. For example,

```
0.0   3.14   +34324.39909   -334.5
```

(b) Exponential Form: In this form, a real constant has two parts—a mantissa and an exponent. The mantissa must be either an integer or a real constant, followed by a letter E or e. The exponent must be an integer constant. For example,

```
152E+03   2.14e-14   0.65E4   -1.2e-1
```

Some invalid real constants are as follows:

`45.`	(Must have at least one digit after decimal point)
`.93`	(Must have at least one digit before decimal point)
`3`	(It is not a real constant because there is no decimal point)
`23,332.34`	(Comma is not allowed)
`34.63.2`	(Two decimal points are not allowed)
`3.4E`	(No digit specified for exponent)
`1.234E5.6`	(Exponent must be an integer constant)

3. Character Constants: A character constant contains a single character enclosed within a pair of single quotes. For example,

```
'A'   'b'   '5'   '0'   '*'   '#'   ' '
```

Note that, in the above examples, last constant is a blank space. Also note that 5 is an integer constant, whereas '5' is a character constant.

Some invalid character constants are:

`'ab'`	(Two characters not allowed between single quotes)
`'\c'`	(It is not a valid escape sequence)

4. String Constants: It is a sequence of one or more characters enclosed between double quotes. These characters may be any character from the 'C' character set. For example,

```
"Akshay"   "Jay\'s book"   "3"   "0"   "34.3e4"   "\nHello"
```

IV. Names/Identifiers

Names, also called identifiers or words, are used to refer the variables, arrays, structures, unions, functions, etc. in the program, whether defined by the programmer or by the 'C'. A 'C' identifier consists of a sequence of alphanumeric characters. The following restrictions or rules apply in defining an identifier:

1. Identifiers may consist of alphabets (both uppercase and uppercase), digits and underscore (_).
2. No other special character is permitted.
3. No commas or blanks are allowed within an identifier.
4. The keywords cannot be used as identifier names.
5. The first letter must be an alphabet or underscore (_). They must not begin with a digit.
6. The first 32 characters (8 characters in some 'C' compilers) are significant, by default. This length can be changed any where between 1 to 32.

Note that, 'C' is a case-sensitive language, that is, upper case and lower case alphabets are different for 'C'. For example, Num and num are two different identifiers for 'C'. So, the case-sensitivity of 'C' is to be kept in mind while defining identifiers.

Some of the valid identifiers are:

```
MAX     Date_Of_Birth     _chk  dayTemperature          MyClass
Num1    MotorCycle sum    Principal_Amount  PI
```

Here are some invalid identifiers:

`Date-Of-Birth`	Hyphen (–) is used
`long`	Keyword is used
`1stNum`	Started with a digit
`Principal Value`	Space is used

What is the purpose of using a data type? Categorise the data types

UNIT

2

Section-A *(Very Short Answer Questions)*

Q.1. Define arithmetic operators. *(2011, 12)*

Ans. 'C' provides all the basic arithmetic operators. Arithmetic operators, which work on two operands are known as binary arithmetic operators, *e.g.*, +, −, *, / and % are arithmetic operators.

Q.2. What do you mean by relational operators?

Ans. We often compare two values and depending on their relation, take certain decisions. If we want to do comparisons in 'C' programs, we can do so with the help of relational operators, *e.g.*, <, <=, >, >=, == and != are called relational operators.

Q.3. What are logical operators? *(2011)*

Ans. 'C' has three logical operators which are &&, II and ! with their meanings as logical AND, logical OR and logical NOT, respectively. Like the relational operators, logical operators also yield values 0 (*i.e.*, true) or non-zero (*i.e.*, false) and they are normally used with some control statements (such as, if, for, while, do...while, etc.)

Q.4. Define bitwise operators.

Ans. 'C' defines six bitwise operators which operate on bit level, *i.e.*, they are used in setting a particular bit or group of bits to 1 or 0. They are also used to perform certain numerical computations faster.

Q.5. What are the three categories of bitwise operators?

Ans. Following are the three categories of bitwise operators:

(a) Bitwise shift operators (>> and <<).

(b) Bitwise logical operators (0, 1 and ^).

(c) One's complement operator (~).

Q.6. What is right-shift operator (>>)?

Ans. Right-shift operator (also known as signed right-shift operator) is a binary operator where the value of the second operand specifies that by how many bit positions the value of the first operand is to be right shifted.

Q.7. What do you mean by control statement?

Ans. There are some situations, where we may have to choose the order of execution of statements (*i.e.* changing the flow of control) based on certain conditions or repeat a group of statements for known number of times or until certain specified conditions are met.

Q.8. Explain the switch statement.

Ans. 'C' provides a multi-way branching statement called switch statement. The switch statement provides a better alternative than a large series of if-else-if statement. The switch statement has more flexibility and a clear format than if-else-if ladder.

Syllabus

Operators : Types of operators, precedence and associativity, expression, statement and types of statements.

Build in Operators and Function: Console based I/O and related built in I/O function: `printf()`; `scanf()`, `getch()`, `getchar()`, `putchar()`; concept of header files, preprocessor directives: `#include`, `#define`.

Q.4. Write short notes on the working of getchar(), getch(), getche() and p... functions. *(2011)*

Or **Differentiate among getch(), getche() and getchar() functions.**

Or **What are unformatted input/output functions in C? Explain.**

Ans. Unformatted Input/Output Functions: These functions only work with the character data type. They do not require conversion symbol for identification of data types as they work only with character data type.

Some of these functions are:

1. The `getchar()` Function: The `getchar()` function returns the character read, after converting it to an int. The typed character will be echoed to the computer screen. After typing the appropriate character, the user is required to press the Enter key.

Syntax: `int getchar(void);`

2. The `getch()` Function: The `getch()` function reads a single character from the keyboard, without echoing it to the screen. This function returns the character read from the keyboard (without waiting for the Enter key to be hit).

Syntax: `int getch();`

3. The `getche()` Function: The `getche()` function reads a single character from the keyboard (without waiting for the Enter key to be hit) and echoes it to the computer screen. This function returns the character read from the keyboard.

Syntax: `int getche(void);`

3. The `putchar()` Function: The `putchar()` function puts the character given by integer c on the output screen. It returns the same character in integer form. The `putch()` function does the same work.

Syntax: `int putchar(int c);`

Section-C *(Long Answer Questions)*

Q.1. Define expression, operands and operators. Categorise the operators based on number of operands.

Or **What are the arithmetic, relational and logical operators? Draw the precedence chart.**

Or **Describe the five arithmetic operators in 'C'. Summarise the rules associated with their use.** *(2011, 12)*

Or **Define operator. Explain various types of operators based upon number of operands. Explain any two one operand operator.** *(2014)*

Or **What is operator? Explain it with its all types in C with complete description.** *(2017)*

Ans. Expression: An expression is a formula for computing a value. It consists of a sequence of operands and operators.

Operands: The operands may contain function calling, variables and constants.

Operators: The operators specify the action to be performed on the operands.

Example:

```
x = 2.0 / 3.0 + a * b;
```

The entire line is a statement (also known as expression statement), but the portion after the equal sign is an expression. =, /, + and * are operators. In the above example, x, 2.0, 3.0, a and b are operands for these operators.

Categories of Operators

1. According to Number of Operands: There are three categories of operators based on number of operands:

(a) **Unary Operators:** Operators, which take only single operand for its operation, are called unary operators. Such as ++, – –, !, etc.

(b) **Binary Operators:** Operators, which take two operands for its operation, are called binary operators. Such as +, /, <, etc.

(c) **Ternary Operators:** Ternary operator takes three operands for its operation. In 'C', only conditional operator is a ternary operator.

2. According to the Function Performed: These are as follows:

I. Arithmetic operators
II. Relational operators
III. Logical operators

I. Arithmetic Operators

'C' provides all the basic arithmetic operators. Arithmetic operators, which work on two operands, are known as binary arithmetic operators. Five binary arithmetic operators supported by 'C' are shown in table 1:

Table 1: Different Binary Arithmetic Operators

Operator	Operation Performed	Works on
+	Addition	Integers, Characters and Floating-points
–	Subtraction	Integers, Characters and Floating-points
*	Multiplication	Integers, Characters and Floating-points
/	Division	Integers, Characters and Floating-points
%	Remainder	Integers and Characters

Note: a % b = a – (a / b)* b
where a and b may be negative or positive. Modulus operator (%) works only on integers.

Examples:

```
12 + 5          results 17
12 - 5          results 7
12 * 5          results 60
12 / 5          results 2 (since the fractional part truncates)
12.0 / 5.0      results 2.4
14 % 3          results 2
 14 % -3        results 2
-14 % 3         results -2
-14 % -3        results -2
```

Apart from these binary operators, + and – also has a unary form (which works only on one operand). So, these two are also known as unary arithmetic operators or one operand operators. Both the operators precede their operand (it is a value on which the operator operates). Both operators work only on integers and floating-point numbers. Their result is also of the same type as that of their operand. Table lists the use and effect of unary + and unary –.

Table 2: Examples of Unary Arithmetic Operators: + and –

Value of Variable m	+m Results	-m Results
34	34	-34
0	0	0
-56	-56	56

II. Relational Operators

We often compare two values and depending on their relation, take certain decisions. For example, we may compare the price of two items, percentage of two students etc. If we want to do these comparisons in 'C' programs, we can do so with the help of relational operators.

'C' provides six relational operators. All the relational operators in 'C' are binary operators, *i.e.* they take two operands to compare them. The result of a relational operator is either zero (boolean false) or non-zero (boolean true).

Knowledge Booster
The binary relational and equality operators compare their first operand to their second operand to test the validity of the specified relationship.

Table 3: Different Relational Operators

Operator	Operation Performed	Works On
<	Is less than	Integers, Floating-point and Characters
<=	Is less than or equal to	Integers, Floating-point and Characters
>	Is greater than	Integers, Floating-point and Characters
>=	Is greater than or equal to	Integers, Floating-point and Characters
==	Is equal to	Boolean, Integers, Floating-point and Characters
!=	Is not equal to	Booleans, Integers, Floating-point and Characters

Examples:

```
4.5 < 7.8       results 1
4.5 > 7.8       results 0
3 >= 5          results 1
5 <= 6-3        results 0
5 == 5          results 1
5 != 5          results 0
```

In the fourth example above, firstly, `6-3` will be evaluated and then its result will be compared with the 5 because, arithmetic operators have higher precedence (*i.e.* priority) than the relational operators.

Also = and == are two different operators. As shown in the table 3, == is a relational operator, whereas = is an assignment operator (used to store some value in a variable).

III. Logical Operators

In addition to the relational operators, 'C' has three logical operators, which are `&&`, `||` and `!`; their meanings are logical `AND`, logical `OR` and logical `NOT`, respectively. Like the relational operators, logical operators also yield values `0` (*i.e.* true) or non-zero (*i.e.* false) and they are normally used with some control statements (such as, `if`, `for`, `while`, `do...while`, etc.).

The logical operators `&&` and `||` are binary operators and are used to form compound conditions by combining two or more relations.

1. **Logical AND (&&) Operator:** The logical AND (&&) operator (also known as short-circuit logical AND operator) results true, if both of its operands are true and false otherwise (Refer the table 4).

Table 4: Working of Logical AND (&&) Operator

Operand 1	Operand 2	Operand 1 && Operand 2
false	false	false
false	true	false
true	false	false
true	true	true

2. **Logical OR (||) Operator:** The logical OR (||) operator (also known as short-circuit logical OR operator) results false, if both its operands are false and true otherwise (Refer the table 5).

Table 5: Working of Logical OR (||) Operator

Operand 1	Operand 2	Operand 1 || Operand 2
false	false	false
false	true	true
true	false	true
true	true	true

3. **Logical NOT (!) Operator:** It is a unary operator. This operator reverses the value of its operand it operates on (Refer the table 6).

Table 6: Working of Logical NOT (!) Operator

Operand 1	! Operand 1
false	true
true	false

Examples:

1. `4 > 3 && (12 != 5)` results true (Both operands are true)
2. `34 < 23 && 12 != 5` results false (Left operand is false and right operand is true)
3. `34 < 23 || 12 != 5` results false (One operand is true)
4. `! (34 < 11 + 12)` results true (Operand is false but unary operator will negate it)

The logical operators && and || have lower precedence than relational operators. So, in the first three examples above, we don't need to enclose the left and right expressions of && and || in parentheses.

But, logical operator ! has higher precedence than relational and arithmetic operators. So, in the last example above, we need to enclose the expression of this operator in parentheses to negate it.

Note: Relational operators and logical operators are also known as boolean operators.

Precedence and Associativity of Operators

The sequence of operations in an expression is governed by what is known as precedence (or hierarchy) of operations. The precedence of an operator determines which operator (in an expression) is to be used first for the execution. Execution of operators of equal precedence is left associative. For example,

```
A = 5 + 3 - 2;
```

In the above statement, the operator + will be evaluated first, which results 8, then the operator – will use this result as the first operand and 2 as the second operand thus giving the final result for the variable A, *i.e.*, 6.

Associativity of an operator may be from left to right or from right to left. It determines the grouping of operands and operators in an expression with more than one operator of the same precedence.

Table summarises the rules for precedence and associativity of all arithmetic, relational and logical operators available in 'C':

Precedence and Associativity of Arithmetic, Relational and Logical Operators

Precedence	Operators	Associativity
1(Highest)	! + -(Unary)	Right to left
2	* / %	Left to right
3	+ -(Binary)	Left to right
4	< <= > >=	Left to right
5	== !=	Left to right
6	&&	Left to right
7(Lowest)	||	Left to right

IV. Bitwise Operators: Refer to Sec-C, Q.2.

V. Assignment Operators: Refer to Sec-B, Q.1.

VI. Special Operators: Refer to Sec-B, Q.2.

This example illustrates a compound statement:

```
if (i > 0)
{   line [i] = x;
    x++;
    i--;
}
```

In this example, if i is greater than 0, all statements inside the compound statement are executed in order.

Q.5. Discuss the input and output statements available in 'C' language alongwith their syntax and usage.

Or **What are formatted I/O statements in 'C'? Explain with example.** *(2012)*

Or **How can you format I/O using scanf() and printf() functions? Explain.** *(2011)*

Ans.

Formatted I/O Statements Functions in C

There are many library functions available for console I/O. Using them in 'C' programs forms input/output statements. The formatted input/output functions read and write all types of data values. They require conversion symbol to identify the data type. Most common input and output functions are—`scanf()` and `printf()`. To use them, the `stdio.h` header file must be included.

I. The scanf() Function

`scanf()` allows us to enter data from keyboard that will be formatted in a certain way.

Syntax: `scanf(format string, list of addresses of variables);`

Here, the format string contains the format specifiers, as shown in the table, that always begin with a % sign.

The list of addresses of variables are used so that `scanf()` function can place the data received from the keyboard. The address of a variable is obtained by using address of operator(&).

The values that are supplied through the keyboard must be separated by either blank(s), tab(s) or newline(s). The escape sequences (such as `'\n'`, `'\t'` etc.) must not be included in the format string. Most commonly used format specifiers used in `scanf()` function are:

Table: List of Format Specifiers

Format Specifiers	Used For
%d	signed int, signed short, signed char
%ld	signed long
%u	unsigned int, unsigned short, unsigned char
%lu	unsigned long
%f	float
%e or %E	float (exponential notation)
%lf	double
%le or %lE	double (exponential notation)
%c	char
%s	String
%%	to print % (used only with printf())

Example: `scanf(%f %f", &x, &z);`

Suppose, input was 12.34 78.90

Then, x will be 12.34 and z will be 78.90.

II. The `printf()` Function

This function outputs different type of data values like characters, string, integers, float, etc. by formatting the data with a string. Its general form looks like this:

Syntax: `printf (format string, list of variables);`

where the format string contains:

1. Characters that are simply printed as they are.
2. Conversion specifications that begin with a % sign, as shown in the table.
3. Escape sequences that begin with a \ sign, such as '\n', '\t', etc.

The `%d, %f,` etc. used in the `printf()` function are called conversion characters.

Example: Consider the following segment of a program:

```
int n = 9876;
float y = 98.7654;
char str[] = "NEW DELHI 110001";
printf("Your account balance is: %d",n);
printf("\nPercentage is: %f",y);
printf("\n%e",y);
```

Output: The output of the program will be:

```
Your account balance is: 9876
Percentage is: 98.765400
                    9.876540e+01
```

UNIT 3

Section-A *(Very Short Answer Questions)*

Q.1. Distinguish `switch` and `for`. *(2014)*

Ans. Switch is a multiway branch statement that provides an easy way to dispatch execution to different parts of code based on the value of the expression.

For statement is an entry controlled loop which is suited for problems where the number of times a statement-block will execute is known in advance.

Q.2. Compare in terms of their functions, `while` and `for` statements.

Ans. Although both `while` and `for` are entry-controlled loops and execute a group of statements repeatedly. There are some differences between them. These differences are shown in the table below:

S.No.	Basis of difference	The `while` Loop	The `for` Loop
1.	Type of loop	It is an indefinite loop.	It is definite loop.
2.	Suitability of loop	This loop is suitable in situations where it is not known in advance that how many times loop will execute.	This loop is suitable in situations where it is known in advance that how many times loop will execute.
3.	Number of control expressions	According to the syntax, it has only one control expression, which evaluates to true or false.	According to the syntax, it has three control expressions. Generally, first expression is used for initialization; second is a condition and third is an updation expression.

Q.3. Write a brief note on the `exit()` function. *(2012)*

Or **Define exit statement.** *(2013)*

Ans. The `exit()` function is declared in the `Stdlib.h` header file. It is used to terminate the currently executing program and returns the integer value n, which represents the exit status of the program. In a Unix-like programming environment, n is usually zero for error-free termination and non-zero for otherwise. The value of n may be used to diagnose the error in UNIX operating system.

Syntax: `void exit (int n);`

Q.4. Explain the use of exit statement in C.

Ans. Exit statement is used to exit the program as a whole. In other words, it returns control to the operating system. After `exit ()` all memory and temporary storage areas are all flushed out and control goes out of program. `Exit ()` statement is placed as the last statement in a program since after this, the program is totally exited.

Example: `exit(0);`

Syllabus

Control Structures: Decisio making structures: if, if-else nested if-else, switch, loop contro structures: while, Do-while, fo nested for loop, other statements break, continue, go to, exit.

Q.5. Distinguish break and continue. *(2011, 14)*

Ans. Differences between break and continue are as follows:

S.No.	Basis of difference	Break	Continue
1.	Appearance	A break can appear in both switch and loop statements.	A continue can appear only in loop statements.
2.	Termination of loop	A break causes the switch or loop statements to terminate the moment it is executed.	A continue doesn't terminate the loop, it causes the loop to go to the next iteration.
3.	Result	A break causes the innermost enclosing loop or switch to be exited immediately.	A continue inside a loop nested within a switch causes the next loop iteration.

Section-B *(Short Answer Questions)*

Q.1. What do you understand by the nested if-else statements? Explain with the help of an example. *(2011)*

Ans. Nested if-else Statements : It is perfectly alright if we write an entire `if-else` construct within either the body of the `if` statement or the body of an `else` statement. There is no limit on how deeply the ifs and elses can be nested. Consider the following program:

Program

```
//Program to check a year is leap or not.
#include <stdio.h>
#include <conio.h>
void main()
{     int y;
      printf("\nEnter a year: ");
      scanf("%d",&y);
      if( y%100 == 0)
      {     if(y%400 == 0)
                  puts("It is a leap year");
            else
                  puts("It is not a leap year");
      }
            else
      {     if(y%4 == 0)
                  puts("It is a leap year");
            else
                  puts("It is not a leap year");
      }
}
```

Output:

```
Enter a year: 2012
It is a leap year.
```

In the given program, entire `if-else` construct is enclosed in the body of the outer `if` statement. Similarly, one more `if-else` construct is enclosed in the body of the outer `else` statement. Thus, there are three nested `if-else`.

Q.2. Distinguish between nested `if` and `switch` statement. *(2011, 12, 13)*

Ans. Differences between nested `if` and `switch` Statements: The switch and if (all forms: simple `if`, `if-else` and `if-else-if` or nested `if`) statements, both are used to select an alternative out of given many alternatives. However, there are some differences between them. These are shown in the table:

S.No.	Nested if Statement	Switch Statement
1.	It can evaluate any type of Boolean expression.	It can only test for equality.
2.	We can also handle ranges.	Since each `case` value must be a single value; so `switch` cannot handle ranges.
3.	Using `if` statements, we can compare integer as well as floating-point values.	Using `switch`, we can only compare `long`, `short`, `int` and `char`-type values.
4.	The condition of `if` may take literal, variable, expression and/or method call.	The `case` value of `switch` must be a literal (*i.e.*, constant).
5.	If a single variable is to be compared with a set of values then, `if` is not so efficient as `switch`.	If a single variable is to be compared with a set of values, then the `switch` is usually more efficient than a set of nested `if` or `if-else-if` ladder.
6.	The if statement doesn't run so faster as `switch` statement, if equivalent logic is coded in both of them.	The `switch` statement run much faster than the equivalent logic coded using a sequence of `if-elses`.

Q.5. Write a short note on the goto statement. (2012)

Ans. goto Statement: In 'C' language, the goto statement is used to transfer the control of a program from one statement to other statement unconditionally. Often goto is used with if statement to act on only certain criteria. We have to insert a label in the program at the desired destination for the goto statement. The label is always terminated by a colon (:). The keyword goto, followed by this label name, then takes the control to the label. Any number of goto statements can take the control to the same label.

Syntax: `goto labelname;`

The label is declared as:

```
labelname:
```

Program

```
// Program to print first five natural numbers.
#include <stdio.h>
#include <conio.h>
void main()
{       int a = 1;
        clrscr();
lab:
        printf("\n%d",a);
        a++;
        if(a <= 5)
        goto lab;
        getch();
}
```

Output:

```
1
2
3
4
5
```

In the above program, lab is the name of the label and the statement goto lab takes the control t the printf() function.

Section-C *(Long Answer Questions)*

Q.1. What is decision making structures? Explain with its suitable example. (201

Or **Explain `if-else` and `nested-if-else` with example.** (201

Ans. Decision Making Structures: Decision making is about deciding the order of execution statements based on certain conditions or repeat a group of statements until certain specifie conditions are met. C language handles decision making by supporting the following statements structures:

1. if statement
2. switch statement

3. conditional operator statement
4. goto statement

Decision making with if statement

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are :

1. Simple if statement
2. if... else statement
3. Nested if... else statement
4. else if statement

1. Simple if statement

The general form of a simple if statement is :

```
if(expression)
{       statement-inside;
}
statement-outside;
```

If the expression is true, then 'statement-inside' will be executed, otherwise 'statement-inside' is skipped and only 'statement-outside' is executed.

Example:

```
#include <stdio.h>
void main()
{       intx,y;
        x = 15;
        y = 13;
        if(x > y)
        {       printf("x is greater than y");
        }
}
```

2. if...else statement

The general form of a simple if...else statement is,

```
if(expression)
{       statement-block1;
}
else
{       statement-block2;
}
```

If the 'expression' is true, the 'statement-block1' is executed, else 'statement-block1' is skipped and 'statement-block2' is executed.

Example:

```
#include <stdio.h>
void main()
{       int x,y;
        x = 15;
        y = 18;
```

```
        if (x > y)
{       printf("x is greater than y");
}
else
{       printf("y is greater than x");
}
}
```

3. Nested `if....else` statement

The general form of a nested `if...else` statement is :

```
if(expression)
        {       if(expression1)
                {       statement-block1;
                }
                else
                {       statement-block2;
                }
        }
        else
        {       statement-block3;
        }
```

If 'expression' is false, the 'statement-block3' will be executed, otherwise it continues to perform the test for 'expression 1'. If the 'expression 1' is true then 'statement-block1' is executed otherwise 'statement-block2' is executed.

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{       int a,b,c;
        clrscr();
        printf("enter 3 numbers");
        scanf("%d%d%d",&a,&b,&c);
        if(a > b)
        {
                if(a > c)
                {
                        printf("a is greatest");
                }
                else
                {
                        printf("c is greatest");
                }
        }
        else
```

```
{
        if(b>c)
        {       printf("b is greatest");
        }
        else
        {       printf("c is greatest");
        }
        }       getch();
}
```

Q.2. Explain the syntax and function of `if` statement (all forms) with suitable example.
Or **Write a program in C to find out the number is even/odd.** *(2016)*
Or **Write a 'C' program to accept number and find out whether it is even or odd.** *(2017)*

Ans. **`If` statement**

There are many forms of `if` statement. We can use any form depending on the complexity of the conditions to be tested. These different forms are:

1. Simple `if` statement
2. The `if-else` statement
3. The `if-else-if` ladder

I. Simple `if` statement

The syntax of simple `if` statement is

Syntax:

```
if (condition)
  Statement-block
```

In the above syntax, `if` is the keyword of 'C'. The condition may represent a relational expression or a logical expression. It is mandatory to enclose the condition of `if` statement in parentheses.

Statement-block may be a single statement or a group of two or more statements. If Statement-block is a single statement, then it is not necessary to enclose this single statement in the braces. But, `if` Statement-block is a group of statements, then it becomes necessary to enclose all these statements in the braces.

The simple `if` statement works like this: If condition evaluates to true, then the statements in the Statement-block (*i.e.*, body of the if statement) are executed and otherwise, this block will be bypassed. This process is illustrated in the Fig. (a).

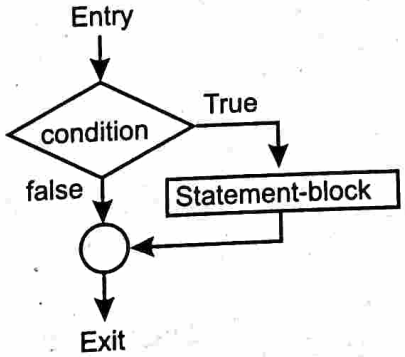


Fig. (a) Flowchart depicting the working of simple `if` statement.

Consider the following examples to understand the working of simple `if` statement:

Example 1 :

```
if (category == SPORT)
{       marks += bonus_marks;
}
```

In the above code, `if` the student belongs to SPORTS category, then additional bonus marks are added to his/her marks and otherwise, bonus marks will not be added.

Example 2 :

```
if (w < 50 && h > 172)
  count++;
```

There may also be a compound condition of the `if` statement. In the above code, if both the conditions are true, then the variable count will be incremented and otherwise, `count` will not be incremented.

Let us see the use of `if` statement in a complete 'C' program. Consider the program 1 below. It uses an `if` statement and check a number is even or odd.

Program 1

```
/*Program to check a number is Even or Odd using a simple if
statement.*/
#include <stdio.h>
#include <conio.h>
void main()
{       int number;
        printf("\nEnter a number: ");
        scanf("%d",&number);
        if(number % 2 == 0)
        {       printf("\nIt is even");
                return;
        }
        printf("\nIt is odd");
}
```

Output:

```
Enter a number: 23
It is Odd
```

Output:

```
Enter a number: 42
It is Even
```

II. The `if-else` Statement

The syntax of `if-else` statement is:

Syntax:

```
if (condition)
   Statement-block-1
else
   Statement-block-2
```

If condition of the `if` statement evaluates to true, then the statements in the Statement-block-1 (*i.e.*, body of the `if` statement) are executed and statement-block-2 is bypassed and otherwise the statements in the Statement-block-2 (*i.e.*, body of the `else` statement) are executed and statement-block-1 is bypassed. This process is also illustrated in the Fig. (b).

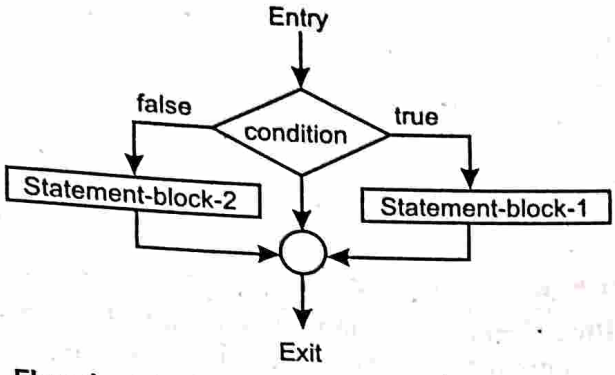


Fig (b). Flowchart depicting the working of `if-else` statement.

Consider the program 2 to understand the working of `if-else` statement. This program is using an `if-else` statement to check that a number is even or odd.

Program 2

```
/*Program to check a number is Even or Odd using a if-else statement.*/
#include <stdio.h>
#include <conio.h>
void main()
{   int number;
    printf("\nEnter a number: ");
    scanf("%d",&number);
    if(number%2 == 0)
        printf("\nIt is even");
    else
        printf("\nIt is odd");
}
```

Output:

```
Enter a number: 24
It is Even
```

Output:

```
Enter a number: 127
It is Odd
```

Note that, if we compare the program 2 (using a simple `if` statement) with program 1 (using an `if-else` statement), both do the same work, *i.e.* checking that a number is even or odd. That means, a single problem may be solved in many ways using different control statements in 'C'.

III. The `if-else-if` Ladder

There is one more way of putting `if`s and `else`s together when multipath decisions are involved. A multipath decision is a chain of `if`s in which an `if` statement is associated with each else. Its syntax is given ahead.

Syntax:

```
if (condition-1)
    Statement-block-1
else if (condition-2)
    Statement-block-2
    :
    :
else if (condition-N)
    Statement-block-N
else
    Statement-block-s
```

This construct is known as the `if-else-if` ladder. The conditions are evaluated in order from top to bottom and if any condition evaluates to true, then the statement block associated with it is executed and this terminates the whole chain. The last `else` part handles the none-of-the-above or default case where none of the specified conditions are satisfied. This sequence of `if` statements is the most general way of writing a multi-way decision.

Consider the program 3 to understand the working of `if-else-if` ladder. This program prints the equivalent day name according to the given day number.

Program 3

```
//Program to print equivalent day name of a user-given day number using if-else-if ladder.*/
#include <stdio.h>
#include <conio.h>
void main()
{     int day_no;
      printf("\nEnter a day number: ");
      scanf("%d",& day_no);
      if (day_no == 1)
            puts("Sunday");
      else if (day_no == 2)
            puts("Monday");
      else if (day_no == 3)
            puts("Tuesday");
      else if (day_no == 4)
            puts("Wednesday");
      else if (day_no == 5)
         puts("Thursday");
      else if (day_no == 6)
            puts("Friday");
      else if (day_no == 7)
            puts("Saturday");
      else
```

```
        puts("Invalid day number");
}
```

Output:

```
Enter a day number: 4
Wednesday
```

Output:

```
Enter a day number: 8
Invalid day number
```

Q.3. Explain the syntax and function of the `switch` statement with suitable example. ***(2013)***

Or **What is the use of `default` statement within a switch statement? What will happen if any `switch` statement does not include the `default` statement?**

Or **Explain `switch` statement.** ***(2017)***

Ans.

The `switch` Statement

Besides `if`, 'C' provides one more multiway branching statement – `switch`. The `switch` statement provides a better alternative than a large series of `if-else-if` statements. The `switch` statement has more flexibility and a clearer format than `if-else-if` ladder. It provides an easy way to dispatch execution to different parts of the code based on the value of an expression. Its syntax is:

Syntax:

```
switch(expression)
{     case val-1:
            statement-block-1
            break;
  case val-2:
            statement-block-2
            break;
             :
             :
  case val-N:
            statement-block-N
            break;
  default:
        statement-block-default
}
```

Knowledge Booster

Switch statement provides an easy way to dispatch execution to different parts of code based on the value of the expression.

In the above `syntax`, `switch`, `case`, `default` and `break` are keywords of 'C'. The expression must be of type `short`, `int`, `long` or `char`. All the case values val-1, val-2, ... , val-N must be a unique constant and of a type compatible with the expression. Duplicate case values are not allowed. The `switch` statement has its own block consists of braces.

The switch statement works like this: If expression takes any value from val-1, val-2, ... , val-N, the control is transferred to that appropriate case. The statements associated with that case are then executed and the `break` statement transfers the control out of `switch` statement. If the value of the expression does not match any of the `case` values, control goes to the `default` case, which is usually at the end of the `switch` statement.

However, the default case is optional. So, if there is no default case, the whole `switch` statement simply terminates when there is no match.

The `break` statement is used inside the `switch` statement to terminate a statement sequence. That is, whenever a `break` statement is encountered, control will immediately exit from the `switch` statement.

We cannot use a `switch` statement to switch on string. Since according to the syntax rules of the 'C' language, all the values after the keyword case must be character- or integer- type, so, we can't specify floating-point or string values there.

Consider the program below, which uses a `switch` statement to find the equivalent day name for a given day number.

Program

```
/* Program to print day name according to the given day number
using switch statement */
#include <stdio.h>
#include <conio.h>
void main()
{     int day_no;
      printf("\nEnter a day number: ");
      scanf ("%d",& day_no);
      switch(day_no)
      {
      case 1:     puts("Sunday");
                  break;
      case 2:     puts("Monday");
                  break;
      case 3:     puts("Tuesday");
                  break;
      case 4:     puts("Wednesday");
                  break;
      case 5:     puts("Thursday");
                  break;
      case 6:     puts("Friday");
                  break;
      case 7:     puts("Saturday");
                  break;
      default:
            puts("Invalid day number");
      }
}
```

Output:

```
Enter a day number: 3
Tuesday
```

Output :

```
Enter a day number: -6
Invalid day number
```

In the above program, suppose the value of variable day_no is 3. First of all, control will compare the value of expression (*i.e.*, day_no) with the first case value (*i.e.*, 1); it doesn't match; now compare with second case value (*i.e.*, 2); once again it doesn't match; again compare with next case value (*i.e.*, 3); it now matched. Thus, control will jump to the statements that followed this case and print the string Wednesday. The next statement break cause the control to exit from the switch statement.

Thus, using break statement in each case is not mandatory but it is optional. If no break statement is used following a case, the control will fall through to the next case. It is sometimes desirable to have multiple cases without break statements between them.

Q.4. Explain the syntax and function of the while and do-while statements with suitable examples.

Or **Give the structure of do-while instruction. Explain how it differ from while instruction.**

Or **Distinguish between while and do-while statements.** ***(2011, 12)***

Or **Explain difference between do-while and while loop with suitable example.** ***(2017)***

Ans.

The while Statement

The while statement is an entry-controlled loop. The while statement is suited for problems where the number of times a statement-block will execute is not known in advance. So, we can also call it an indefinite loop. Its syntax is:

Syntax:
```
while (expression)
    Statement-block
```

Here, expression is a Boolean expression. Statement-block may be a single statement or a group of two or more statements.

The while statement works like this: Firstly, control jumps to expression and evaluates it. If it is true, control jumps to Statement-block and execute all its statements. Now, the control again jumps to the expression and evaluates it. The Statement-block executes repeatedly till the expression evaluates to true. Whenever expression evaluates to false, the control exits the while statement. Fig. (a) depicts the working of the while loop:

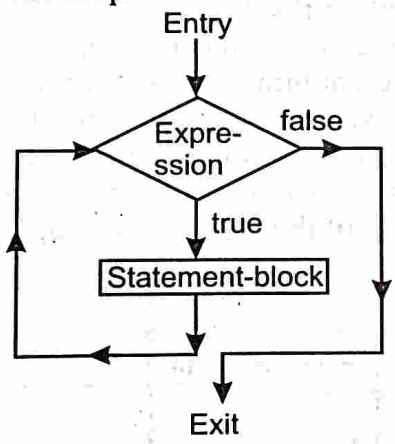


Fig. (a) Flowchart depicting the working of while statement.

Let us consider a program in which it is not confirmed during the writing of this program that how many times the loop will execute. Program 1 calculates the sum of digits of a given number using the definite loop: while.

Program 1

```
/* Program to print sum of digits of an user-given number using
the while loop*/
```

```
#include <stdio.h>
#include <conio.h>
void main()
{       long num;
        int a,s = 0;
        printf("\nEnter a number: ");
        scanf("%ld",&num);
        while( num > 0 )
        {       a = num % 10;
                num = num / 10;
                s = s + a;
        }
        printf("\nIts digits' sum is: %d",s);
}
```

Output:

```
Enter a number: 347
Its digits' sum is: 14
```

The do-while Statement

Unlike the `for` and `while` loops, the `do-while` is an exit-controlled loop. This means that a `do-while` loop always executes at least once. The `do-while` statement is suited for problems where the number of times a statement-block will execute is not known in advance. So, we can also call it an indefinite loop. Its syntax is:

Syntax:

```
do
{ Statement-block
} while( expression );
```

Here, expression is a Boolean expression. Notice that, it is mandatory to enclose all the statements of the Statement-block within the braces.

The `do-while` statement works like this: Firstly, control jumps to Statement-block and executes all its statements. Then, control jumps to expression and evaluates it. If it is true, control again jumps to Statement-block and executes all its statements. Now, the control again jumps to the expression and evaluates it. Thus, the Statement-block executes repeatedly till the expression evaluates to true. Whenever expression evaluates to false, the control exits from the `do-while` statement. Fig. (b) depicts the working of the `do-while` loop.

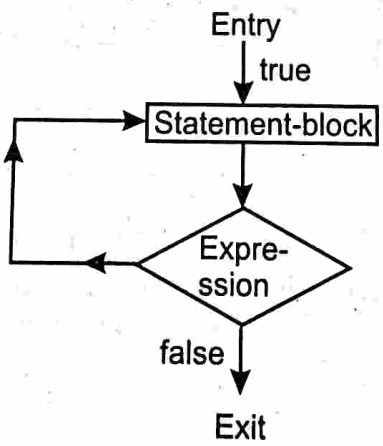


Fig. (b) Flowchart depicting the working of `do-while` *statement.*

Consider the program 2 to understand the working of do-while loop.

Program 2

```
/* Program to print sum of digits of an user-given number using
the do-while loop*/
#include <stdio.h>
#include <conio.h>
void main()
{
      long num;
      int a,s = 0;
      printf("\nEnter a number: ");
      scanf("%ld",&num);
      do
      {     a = num % 10;
            num = num / 10;
            s = s + a;
      }     while( num > 0 );
      printf("\nIts digits' sum is: %d",s);
}
```

Output:

```
Enter a number: 352
Its digits' sum is: 10
```

The while and do-while statements, both come in the category of definite loop. However, there are some differences between them. These are shown in the table:

S.No.	The while Loop	The do-while Loop
1.	It is an entry-controlled loop.	It is an exit-controlled loop.
2.	Only one keyword (while) is used.	Two keywords (do and while) are used.
3.	Braces are not mandatory if there is only one statement in the body of the loop.	Braces are mandatory whether there are one or more statements in the body of the loop.
4.	There is no semicolon after the while condition.	Semicolon is mandatory after the while condition.
5.	If condition becomes false in the first attempt, then body of the loop will not execute at all.	If condition becomes false in the first attempt, then body of the loop will execute at least once.

Q.5. Explain the syntax and function of the for statement with suitable examples.
Or **Explain any one entry control loop with its syntax.** ***(2013)***

Ans.

The for Statement

The for statement is an entry-controlled loop. The for statement is suited for problems where the number of times a statement-block will execute is known in advance. So, we can also call it a definite loop. Its syntax is:

Syntax: `for(expression1 ;expression2 ;expression3)`

Statement-block

In the above syntax, `for` is the 'C' keyword. It is followed by three expressions separated by semicolon. It is mandatory to enclose all the expressions of `for` statement in parentheses.

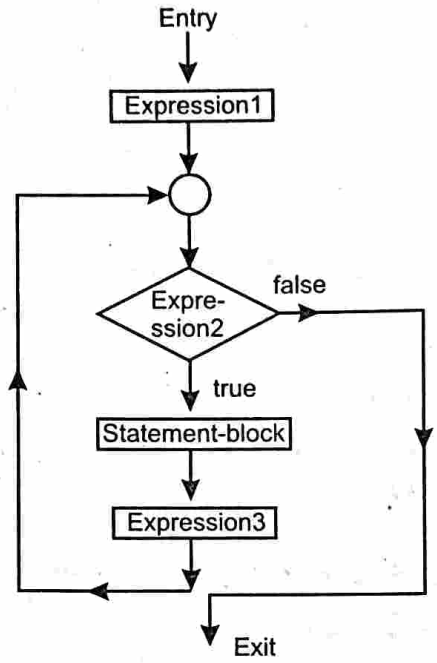


Fig. Flowchart depicting the logic of `for` statement.

Statement-block may be a single statement or a group of two or more statements. If Statement-block is a single statement, then it is not necessary to enclose this single statement in the braces. But, if Statement-block is a group of statements, then it becomes necessary to enclose all these statements in the braces.

The `for` statement works like this: Firstly, control jumps to expression1 and execute it, then control jumps to expression2. If expression2 evaluates to true, control jumps to Statement- block and executes all the statements of it and then control jumps to expression3 and executes it. Now, again control jumps to the expression2 and evaluates it. This process continues till the expression2 evaluates to true. If expression2 evaluates to false, the control exits from the `for` statement. The working of the `for` loop is depicted in the figure.

Generally, expression1 is an initialization expression, expression2 is a Boolean expression and expression3 is an updation expression. So, the syntax of the `for` statement can also be treated like this:

Syntax:

```
for(initialization ; condition ; updation)
    Statement-block
```

In most of the programs, a variable is used to control the working of `for` loop. It is initialized by an initial value in the initialization section, a test condition is prepared on the same variable in the condition section and increment or decrement is done on the same variable in the updation section. So, the `for` loop is controlled with the help of a variable. This variable is generally known as the control variable.

To understand the working of the for loop, consider the following examples:

Example 1:

```
for (a = 1; a <= 5 ; a++)
{     printf("Hello India!");
}
```

The above for loop will execute five times and so prints the message Hello India! five times. In the above code, firstly the control variable a initializes by the value 1 (*i.e.*, execution of the initialization section); then control will check whether a is less than or equal to 5 or not (*i.e.*, evaluation of the condition in the condition section). At this time, this condition is true, so control will executes the printf() function (*i.e.*, the body of the loop). Now, control will increment a (*i.e.*, the execution of updation section) and again evaluates the condition. This process will be continued till the condition is true. As soon as value of a becomes 6, the condition becomes false and control exit from the for loop.

Example 2:

```
for (O = 5 ; O >= 1 ; O--)
      printf("\n%d", O);
```

The above for loop will also execute five times, but the output would be a series from 5 to 1.

Example 3:

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

The printf() function in the above for loop will never be executed because the condition fails at the very beginning itself.

Consider the program, which prints the table of a given number using the for loop.

Program

```
/* Program to print the table of a number using the for loop */
#include <stdio.h>
#include <conio.h>
void main()
{     int a, n;
      printf("\nEnter a number: ");
      scanf("%d",&n);
      printf("Table of %d", n);
      printf("\n==========");
      for(a = 1; a <= 10 ; a++)
      printf("\n%d × %d = %d",n,a,n*a);
}
```

Output:

```
Enter a number: 5
Table of 5
==========
5 ×  1 = 5
```

Knowledge Booster

The for statement lets us specify the initialisation, test and update operations of a structured loop in a single statement.

```
5 × 2 = 10
5 × 3 = 15
5 × 4 = 20
5 × 5 = 25
5 × 6 = 30
5 × 7 = 35
5 × 8 = 40
5 × 9 = 45
5 × 10 = 50
```

Q.6. What are jumping statements? Explain `break` and `continue` statements.

Or **Explain the syntax and function of `continue` statement with suitable example.**

Or **Compare in terms of their functions `break` and `continue` statements.** *(2012)*

Or **Distinguish between `break` and `continue` statement.** *(2017)*

Ans.

Jumping Statements

The process of unconditionally transferring control from one point to another point in the program is called jumping. 'C' supports three jumping statements—`return`, `break` and `continue`.

Using the `break` Statement

The `break` statement transfers the control out of the block in which it is used. The break statement must be used in conjunction with a `for`, a `while`, a `do-while` and a `switch` statement.

When `break` statement executes in the `switch` statement, it terminates the statement sequence. When `break` statement executes in a loop, control exits the loop immediately. It bypasses the condition and any remaining code in the body of the loop.

The `break` statement is an unconditional branching statement. Practically, an unconditional `break` statement has no use. So, we should use this statement with the `if` statement. Working of a `break` statement in `for`, `while` and `do-while` loop is shown in the Fig. (a).

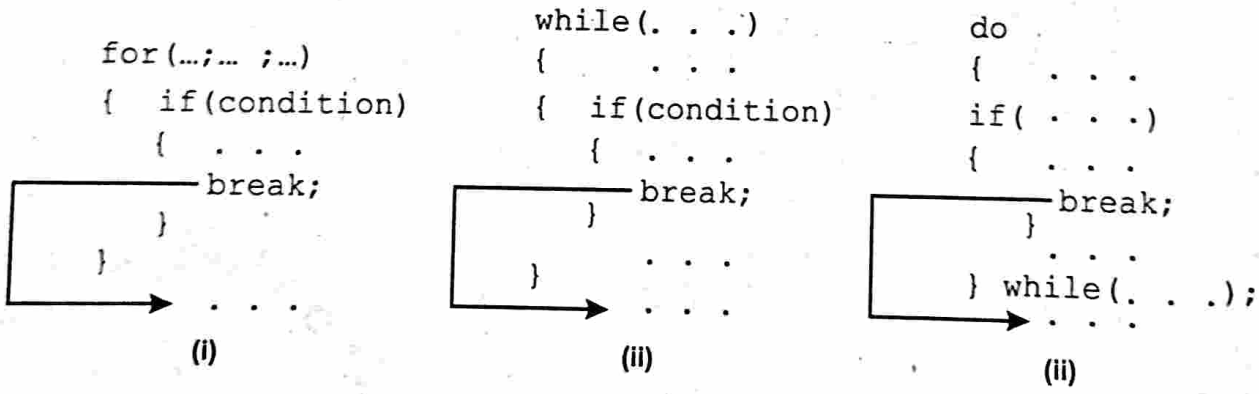


Fig. (a) Working of break statement in (i) `for` (ii) `while` and (iii) `do-while` loops.

To understand the working of `break` statement in the for loop, consider the program 1, which checks that a number is prime or not.

Program 1

```
/* Program to check that a given number is prime or not*/
#include <stdio.h>
#include <conio.h>
void main()
 {    int n,a,flag=1;
      printf("\nEnter a number: ");
      scanf("%d",&n);
      for(a = 2; a <= n/2; a++)
      {     if( n % a == 0)
            {     flag = 0;
                  break;
            }
      }
      if(flag == 1)
      printf("%d is a prime number", n);
      else
      printf("%d is not a prime number", n);
}
```

Output:

```
Enter a number: 5
5 is a prime number
```

Output:

```
Enter a number: 9
9 is not a prime number
```

In the above program, whenever the `if` condition inside the `for` loop evaluates to true, `flag` variable initializes to `0` and `break` will execute, which causes the `for` loop to be terminated immediately.

Using the `continue` Statement

In looping statements, sometimes a situation may arise where we want that from a given statement onwards up to the last statement of the loop are to be bypassed. This task is accomplished by using the continue statement.

The `continue` statement forces the next iteration of the loop to take place and bypass the statements which are not yet executed. The `continue` statement must be used in conjunction with a `for`, a `while` and a `do-while` statement.

Like the `break` statement, the `continue` statement is an unconditional branching statement. Practically, an unconditional continue statement have no use. So, we should use this statement with the `if` statement.

In case of `for` loop, `continue` statement causes the control to be jumped to updation section. In case of `while` and `do-while` loop, `continue` statement causes the control to be jumped to the condition. Working of a `continue` statement in `for`, `while` and `do-while` loop is shown in the Fig. (b).

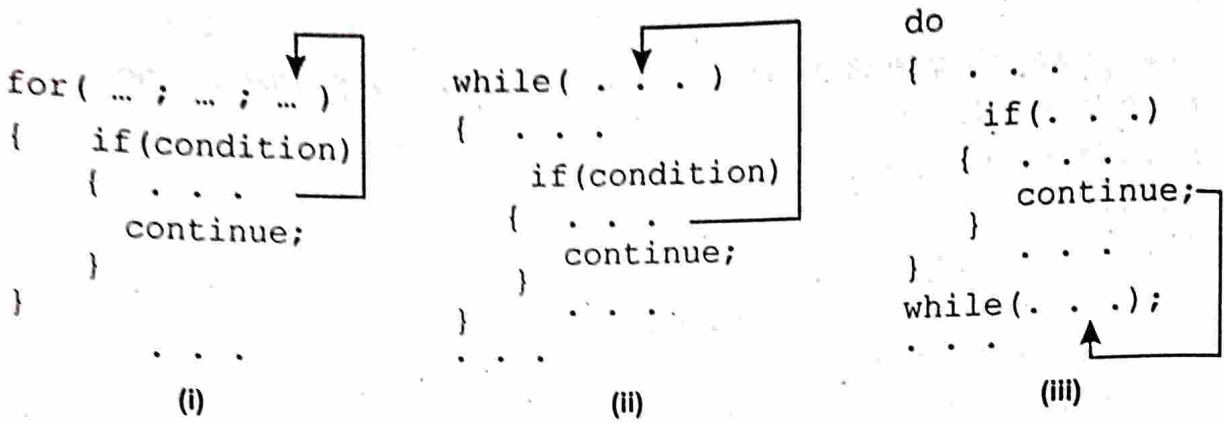


Fig. (b) Working of `continue` statement in (i) `for` (ii) `while` and (iii) `do-while` loops.

Suppose, we want to print the series of numbers from 1 to 20 except 9. One logic may say that run two `for` loops—one from 1 to 8 and other loop from 10 to 20. But as shown in the program 2, it has only one `for` loop from 1 to 20 and in its body the `printf()` function will be skipped for number 9 using the `continue` statement.

Program 2

```
/*Program to print a series of numbers from 1 to 20, but not to
print 9 */
#include <stdio.h>
#include <conio.h>
void main()
{   int a;
    for(a = 1;a <= 20;← a++)
    {   if(a == 9)
        {   continue;
        }
        printf("%d ", a);
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20
```

UNIT

4

Syllabus

Introduction to Problem Solving Concept: Problem solving, problem solving techniques (trial & error, brainstorming, divide & conquer).

Steps in problem solving (Define problem, analyse problem, explore solution). Algorithms and flow-charts (definitions, symbols), characteristics of an algorithm, conditionals in pseudo-code, loops in pseudo code.

Time complexity: Big-O notation, efficiency.

Simple Examples: Algorithms and flowcharts (real life examples).

Section-A (Very Short Answer Questions)

Q.1. Define a program.

Ans. A computer acts on the basis of information and program fed into it. The computer does not have a mind of its own. "A specification of a sequence of computation steps (or instructions) telling the computer 'What to do', in a particular programming language is referred to as program."

Q.2. What is concurrent programming?

Ans. Concurrent programming refers to the design and development of programs for parallel execution of several processes/tasks. Such programs are used in multi-user operating systems, multiprocessing systems and computer networks.

Q.3. What do you meant by problem-solving?

Ans. Problem-solving is a mental process and is a part of the larger problem process that includes problem finding and problem shaping. Problem solving includes the ability to recognise and define problems, invent and implement solutions and track and evaluate results.

Q.4. Give any four characteristics of a difficult problem.

Ans. The characteristics of difficult problem are as follows:

(a) Complexity (large number of items, interrelations and decisions).

(b) Intransparency (lack of clarity of the situation).

(c) Dynamics (time considerations).

(d) Polytely (multiple goals).

Q.5. What are the various problem solving techniques?

Ans. The various problem solving techniques are as follows:

(a) Trial and error.
(b) Root cause analysis.
(c) Divide and conquer.
(d) Research.
(e) Analogy.
(f) Lateral thinking.

Q.6. Describe the relevance and importance of problem solving techniques. *(2013)*

Ans. Problem solving is a mental process. It gives us ability to recognise and define problems. Problem solving on computers provide us solution of the problem in terms of simple concepts, operations and computer code.

Q.7. What do you understand by an algorithm? *(2015)*

Ans. Algorithm is a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired results will be obtained.

Q.16. Write an algorithm for the following problem:
'Interchange the value of two integer type variables with each other.'

Sol. Algorithm Name: INTERCHANGE. Arguments: INT A and INT B.
Step I: Let TEMP = INT A.
Step II: Let INT A = INT B.
Step III: Let INT B = TEMP.
Step IV: Stop.

Q.17. What do you mean by debugging? *(2012)*

Ans. As long as human beings make computer programs they will have errors. Program errors are known as bugs and the process of detecting and correcting those errors is called debugging.

Section-B *(Short Answer Questions)*

Q.1. What is divide and conquer technique to solve a problem? Explain it by giving an example. *(2011)*

Ans. Divide and Conquer Method: In general, mostly the algorithms are recursive in nature, *i.e.* they call themselves repeatedly several times. These algorithms come under the category of divide and conquer method. Here, the entire problem is divided into several subproblems. The subproblems are similar to original problems but are smaller in size. These subproblems are then, in turn, solved recursively. All the solutions of subproblems are combined to get the solution of original problem. Each problem or subproblem is solved using three steps:

1. **Divide:** The problem is divided into several subproblems.
2. **Conquer:** The problems are solved recursively.
3. **Combine:** Here, the solutions of subproblems are combined to get the original solution.

Example:

Divide: The array $A\ [P..r]$ are divided into two subarrays $A\ [P.... q-1]$ and $A\ [q+1..... r]$ such that each element of $A\ [P..... q-1]$ is less than or equal to $A\ [q]$ and $A\ [q+1 ..... r]$ is greater than $A\ [q]$.

Q.2. Write the short note on brainstorming. *(2013)*

Or **What is brainstorming technique? Explain with its example.** *(2015)*

Ans. Brainstorming: Brainstorming is a process of developing creative solutions to problems. Alex Faickney Osborn, an advertising manager, popularised the method in 1953 in his book, Applied Imagination. Ten years later, he proposed that teams could double their creative output with brainstorming.

Brainstorming works by focusing on a problem, and then deliberately coming up with as many solutions as possible and by pushing the ideas as far as possible. One of the reasons why it is so effective is that the brainstormers not only come up with new ideas in a session, but also spark off from associations with other people's ideas by developing and refining them.

While some research has found brainstorming to be ineffective, this seems more of a problem with the research itself than with the brainstorming tool.

Basic Rules of Brainstorming

There are four basic rules in brainstorming which are intended to reduce social inhibitions among team members, stimulate idea generation, and increase overall creativity. These are:

1. No Criticism: Criticism of ideas are withheld during the brainstorming session as the purpose is on generating varied and unusual ideals and extending or adding to these ideas. Criticism is reserved for the evaluation stage of the process. This allows the members to feel comfortable with the idea of generating unusual ideas.

2. Welcome Unusual Ideas: Unusual ideas are welcomed as it is normally easier to 'tame down' than to 'tame up' as new ways of thinking and looking at the world may provide better solutions.

3. Quantity Wanted: The greater the number of ideas generated, the greater the chance of producing a radical and effective solution.

4. Combine and Improve Ideas: Not only are a variety of ideals wanted, but also ways to combine ideas in order to make them better.

Q.3. What do you understand by time complexity of an algorithm? *(2015)*

Or **Why time complexity is an important issue? Explain.** *(2014, 16)*

Or **Explain complexity measures. What is time complexity?**

Or **Define time complexity.** *(2017)*

Ans. Complexity Measures: As an algorithm is a sequence of steps to solve a problem, there may be more than one algorithm to solve the same problem. For example, there are two methods available for searching an element in the list of an array—Linear search and Binary search.

We should choose an algorithm that is more efficient than others. The choice of a particular algorithm depends on the following considerations:

1. Time complexity (or Performance requirements).
2. Space complexity (or Memory requirements).

Specifically, it is used to help in determining how the resource requirements of an algorithm grow in relation to the size of the data being manipulated. Performance requirements are usually more critical than memory requirements.

Knowledge Booster

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time Complexity

The time complexity of an algorithm is the amount of time it needs to run to completion. Time complexity is an important issue because of the following reasons:

1. When we are interested to know in advance that whether the program will provide a satisfactory real-time response.
2. There may be several possible solutions with different time requirements.

We are interested to estimate the execution time of an algorithm irrespective of the computer on which it will be used. This is so because, speed of different computers vary due to the hardware and software used in them.

So, the more reasonable approach is to identify the key operation and count such operations performed till the program completes its execution. A key operation in the algorithm is an operation that takes maximum time among all possible operations in the algorithm. The time complexity can now be expressed as a function of number of key operations performed.

For example, in a selection sort algorithm, the comparison operation may be chosen as the key operation.

Q.4. What do you understand by the pseudocode? *(2013)*

Ans. Pseudocode: Pseudocode is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and subroutines. The programming language is augmented with natural language descriptions of the details, where convenient, or with compact mathematical notation.

The purpose of using pseudocode is that it is easier for humans to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer program development, for sketching out the structure of the program before the actual coding takes place.

No standard for pseudocode syntax exists, as a program in pseudocode is not an executable program. Pseudocode resembles, but should not be confused with skeleton programs including dummy code, which can be compiled without errors. Flowcharts and UML charts can be thought of as a graphical alternative to pseudocode, but are more spacious on paper.

Textbooks and scientific publications related to computer science and numerical computation often use pseudocode in description of algorithms, so that all programmers can understand them, even if they do not all know the same programming languages.

Pseudocode generally does not actually obey the syntax rules of any particular language. There is no systematic standard form, although any particular writer will generally borrow style and syntax like control structures from some conventional programming language. For example, Pascal style pseudocode

```
IF <condition>
     DO action;
ELSE
     DO next action;
```

Q.6. Write an algorithm to reverse the given digit.

Ans. Algorithm to Reverse Digits of a Number:

Input num

1. Initialise rev_num = 0
2. Loop while num > 0
   - Multiply rev_num by 10 and add remainder of num divide by 10 to rev_num
     rev_num = rev_num*10 + num%10;
   - Divide num by 10
3. Return rev_num

Example:

num = 4562
rev_num = 0
rev_num = rev_num *10 + num%10 = 2
num = num/10 = 456
rev_num = rev_num *10 + num%10 = 20 + 6 = 26
num = num/10 = 45
rev_num = rev_num *10 + num%10 = 260 + 5 = 265
num = num/10 = 4
rev_num = rev_num *10 + num%10 = 2650 + 4 = 2654
num = num/10 = 0

Section-C *(Long Answer Questions)*

Q.1. Discuss the different techniques for problem solving with the help of examples.

Or **What do you understand by problem solving techniques? Explain trial and error technique with example.** *(2014)*

Or **Write a short note on—Trial and Error.** *(2015)*

Or **What do you understand by problem solving techniques? Explain divide and conquer.** *(2016)*

Ans.

Problem Solving

Problem solving is a mental process and is part of the larger problem process that includes problem finding and problem shaping.

Problem solving skills include the ability to recognise and define problems, invent and implement solutions and track and evaluate results. Each of us is capable of solving problems in our own way. However, each of us can work to improve our problem solving skills. Algorithms, diagramming and programming are all tools in the process of problem solving in computer science.

Problem solving on computers is the task of expressing the solution of the problem in terms of simple concepts, operations and computer code. The process of designing and writing a program can be subdivided into a number of tasks such as:

1. Understanding the problem.
2. Producing an algorithm to solve the problem.
3. Translating the algorithm into a specific programming language.
4. Testing the resultant program.
5. Iterating the program until the program is correct.

Characteristics of Difficult Problems

As explained by Dietrich Dorner and later expanded upon by Joachim Funke, difficult problems have some typical characteristics that can be summarised as follows:

1. Complexity (*i.e.*, large number of items, interrelations and decisions)
   (a) enumerability.

(b) connectivity (hierarchy relation, communication relation and allocation relation).
(c) heterogeneity.

2. Intransparency (*i.e.*, lack of clarity of the situation)
   (a) commencement uncertainity.
   (b) continuation uncertainty.
3. Dynamics (*i.e.*, time considerations)
   (a) temporal constraints.
   (b) temporal sensitivity.
   (c) phase effects.
   (d) dynamic unpredictability.
4. Polytely (*i.e.*, multiple goals)
   (a) inexpressiveness.
   (b) opposition.
   (c) transience.

The resolution of difficult problems requires a direct attack on each of these characteristics that are encountered.

Problem Solving Techniques

The different techniques for problem solving are as follows:

1. Trial and Error: Testing possible solutions until the right one is found.

2. Root Cause Analysis: Eliminating the cause of the problem.

3. Divide and Conquer: Breaking down a large, complex problem into smaller, solvable problems.

4. Research: Employing existing ideas or adapting existing solutions to similar problems.

5. Reduction: Transforming the problem into another problem for which solutions exist.

6. Brainstorming: Especially among groups of people suggesting a large number of solutions or ideas and combining and developing them until an optimum is found.

7. Analogy: Using a solution that solved an analogous problem.

8. Hypothesis Testing: Assuming a possible explanation to the problem and trying to prove (or in some contexts, disprove) the assumption.

9. Abstraction: Solving the problem in a model of the system before applying it to the real system.

10. Lateral Thinking: Approaching solutions indirectly and creatively.

11. Means-ends Analysis: Choosing an action at each step to move closer to the goal.

12. Method of Focal Objects: Synthesizing seemingly non-matching characteristics of different objects into something new.

13. Morphological Analysis: Assessing the output and interactions of an entire system.

Divide and Conquer: Refer to Sec-B, Q.1.

Q.2. Explain various steps involved in problem solving. (201

Ans.

Steps in Problem Solving

Problem solving on computers is the task of expressing the solution of the problem in terms simple concepts, operations and computer code. The newly developed system designed to replace th current system is known as candidate system. The development of a candidate system involves thre main stages. These stages are explained are as follows:

I. Define the Problem

One must know the exact problem, then he/she can start to solve it. The basis for a candidate system is recognition of a need for improving an information system. For example, a supervisor may want to investigate the system flow in purchasing. This need leads to a preliminary survey or an initial investigation to determine whether an alternative system can solve the problem. It involves looking into the duplication of effort, bottlenecks, inefficient existing procedures or whether parts of the existing system would be candidates for computerisation.

If the problem is more serious, management may want to have an analyst to look at the problem. Such an assignment of work implies a commitment between the management and the analyst. Now, the analyst prepares a statement specifying the scope and objective of the problem. Then, the analyst reviews it with the user (*i.e.*, management) to check accuracy. At this stage, only a rough estimate of the development cost of the project may be reached. However, the accurate development cost may be calculated at the next phase – the feasibility study.

II. Analyze the Problem

Analysis is a problem in which the various operations performed by a system are studied in detail. Their working within and outside of the system is also studied. A key question during the analysis is: What must be done to solve the problem?

One aspect of analysis is to define the boundaries of the system and to determine whether or not a candidate system should consider other related systems.

System analyst collects data available in different files. He/She also investigates various transactions handled by the present system. Analyst uses some logical system models and tools during the analysis, such as, data flow diagrams, interviews, questionnaires, on-site observations, etc. Training, experience and common-sense are required for collection of the information needed to do the analysis.

Once analysis of the system is completed, the analyst has a firm understanding of what is to be done. Now, next step is to explore the solution for the proposed system.

III. Explore the Solution

Design is the most creative and challenging phase of the system life cycle. It describes a final system and the process by which it is developed. The key question here is:

How should the problem be solved?

The first step of design is to determine how the output is to be produced and in what format. Second, input data and master files (database) have to be designed to meet the requirements of the proposed output. The processing phases are handled through program construction and testing. It involves a list of programs needed to meet the system's objectives. Finally, the documentation is done, which is to be evaluated by the management, so that, implementation phase can be started.

The documentation after designing the system includes procedures, flowcharts, record layout, report layouts and a workable plan for implementing the candidate system.

Q.3. What are the five characteristics of an algorithm? ***(2012)***

Or **What is algorithm? What are its essential properties and characteristics? Write the steps involved in developing an algorithm.**

Or **Explain algorithm along with its advantages and disadvantages.**

Or **What is algorithm? Explain.** ***(2017)***

Ans.

Algorithm

'Algorithm is a logical process of analysing a mathematical problem and data step-by-step that can be carried out by a computer or for conversion of data into information'.

Or 'Algorithm is a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired results will be obtained'.

Certain operations must be performed on data for producing the required results. In our day-to-day work, our brain automatically performs the different algorithms by experience (known as heuristic) but not in an error-free and systematic way. But one has to be systematic and to the point (one must use algorithm) for processing on computer because the computer does not know alternatives. For example, we must read a record before using its content in calculations. Sometimes, it is necessary to perform an operation only if the data satisfies a particular condition. For example, age of an employee must be greater than or equal to 18 years.

There may be more than one strategy (or algorithm), for solving a given problem. The selection of a particular algorithm is based upon some elements such as type and availability of data, time and cost factors and availability of equipments.

Example

Let's say that we have a friend arriving at the airport and our friend needs to get from the airport to our house. Here are three different algorithms that we might give our friend for getting to our home:

1. **The taxi algorithm:**
   (a) Go to the taxi stand.
   (b) Get in a taxi.
   (c) Give the driver my address.
2. **The call-me algorithm:**
   (a) When the plane arrives, call on cell phone.
   (b) Meet me outside baggage claim.
3. **The bus algorithm:**
   (a) Outside baggage claim, catch bus number 564.
   (b) Transfer to bus 532 on main street.
   (c) Get off on Gandhiji marg street.
   (d) Walk two blocks north to my house.

All three of these algorithms accomplish exactly the same goal, but each algorithm does it in a completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. So, the algorithm is chosen based on the circumstances.

In computer programming, there are often many different ways - algorithms - to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. Sorting is one place where a lot of research has been done, because computers spend a lot of time in sorting lists. There are different algorithms that are used in sorting—Selection sort, Merge sort, Bubble sort, Insertion sort, etc.

Essential Properties of an Algorithm

Some essential properties of an algorithm are:

1. It should be simple.
2. It should involve a finite number of steps to arrive at a solution.
3. It should lead to a unique solution of the problem.
4. It should be clear with no ambiguity.
5. It should have the capability to handle some unexpected situations, which may arise during the solution of a problem (*e.g.*, division by zero).

Characteristics of Algorithm

The algorithm has following five basic characteristics:

1. **Input:** Any algorithm starts with certain statements to accept the input. The next set of statements would be regarding the process method, which are unambiguous and definite.
2. **Process Method:** The processing method in an algorithm must be very clear and precise and should be workable.
3. **Finiteness:** An algorithm should have definite steps and after executing all these steps, it should come to an end. It should never get into an infinite loop.
4. **Effective Basic Instructions:** Every step in an algorithm must be executed. This means that all operations to be performed must be very basic that they can be executed within a definite period of time.
5. **Output:** An algorithm must produce an output or more than one outputs.

Some other features are:

(a) Simplicity of logic.
(b) Efficiency with which it can be executed.
(c) Adaptability on a computer.

Steps Involved in Developing an Algorithm

The various steps used to develop an algorithm are:

1. Properly understand the problem so that a proper algorithm can be evolved.
2. Study the outputs to be generated from the algorithm so that the input can be specified.
3. Design the process, which will produce the desired result after taking the required input data.
4. Refine the process.
5. Test the algorithm by giving the test data (input data) and see the output. If the desired output is not generated, make appropriate changes in the algorithm and repeat the process.

Advantages and Disadvantages of Algorithm

Advantages: Some of the advantages of an algorithm are:

1. It is a step-by-step solution for a given problem, which is easy to understand for everyone.
2. It has a definite procedure, which can be executed within a set period of time.
3. It is easy to first develop an algorithm for a problem and then convert it into a flowchart and then into a computer program in a desired programming language.
4. It is easy to debug.
5. It is independent of programming languages.
6. There are definite procedures to produce output(s) within a specified period of time.

Disadvantage: It is cumbersome and time consuming (for programmers), as an algorithm is developed first which is converted into a flowchart and then into a computer program in a desired programming language.

Example

Develop an algorithm to convert an integer numerical score (0 to 100) scored by a student in a particular test into letter grades using the following procedure:

Numerical Score	Letter Grade
Less than 33	E
33 to 44	D
45 to 59	C
60 to 74	B
More than 74	A

Sol. Algorithm is defined as follows:
Step I: INPUT the score of a student.
Step II: If the score is less than 33, then PRINT "E": END.
Step III: If the score is greater than or equal to 33 and less than 45, then PRINT "D": END.

Knowledge Booster
An algorithm is a self-contained step-by-step set of operations to be performed to solve a specific problem or a class of operations.

Step IV: If the score is greater than or equal to 45 and less than 60, then PRINT "C": END.
Step V: If the score is greater than or equal to 60 and less than 75, then PRINT "B": END.
Step VI: If the score is greater than or equal to75, then PRINT "A".
Step VII: End of a program.

The above algorithm terminates after 7 steps, which explains the feature of finiteness. Action of each step is precisely defined. In this example, each step requires simple comparison and printing operation. This explains the feature of definiteness and effectiveness. Input of above algorithm is marks scored by a student and output is the grade awarded according to the given range.

Q.4. What is flowchart? Explain suitable example and symbols. *(2015)*

Or **What is a flowchart? How does it help in program development? Write the general guidelines for preparing a flowchart. Give their pictorial representation.**

Or **Write the advantages and limitations of using flowcharts. Draw a flowchart to calculate the simple interest.**

Or **What is a flowchart? Explain its symbols by designing a flowchart for finding whether given number is prime or not.** *(2014)*

Or **What is flowchart? Draw a flowchart to find the largest of given three numbers.** *(2011,13)*

Or **What is flowchart? What are the various symbols used in drawing flowchart?** *(2017)*

Ans.

Flowchart

A flowchart is a step-by-step diagrammatic or pictorial representation of the algorithm for the solution of a problem and flow-charting is the technique of drawing the flowchart.

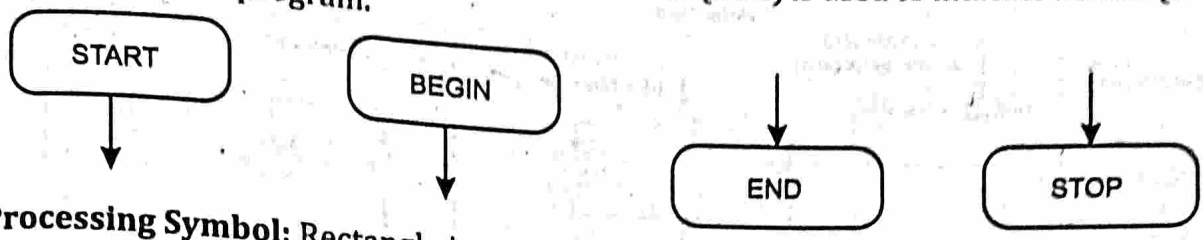
Flowchart helps in Program Development: It helps a person to understand the sequence of steps necessary to solve a given problem at a glance. Once developed and properly checked, the flowchart may serve as an excellent guide for writing the program.

Flowchart Symbols

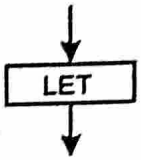
Flowcharts are constructed by using special geometrical symbols. Each symbol represents an activity. The activity could be input/output of data, processing of data, taking a decision, terminating the solution, etc. Arrows joins the symbols to obtain a complete flowchart. Designs of flowchart are very much flexible, so no two people can draw the exact similar flowchart of a single problem.

The symbols described below are as per conventions followed by ISO (International Standards Organisation).

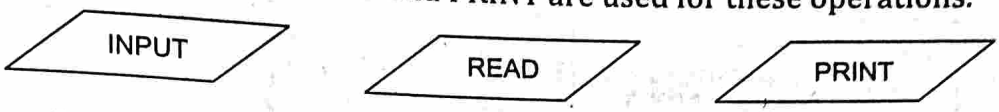
1. Terminal Symbol: Rectangle with rounded sides (oval) is used to indicate BEGIN (or START) and END (or STOP) of a program.



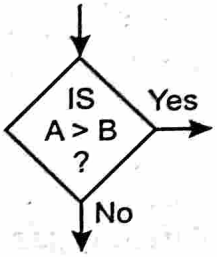
2. Processing Symbol: Rectangle is used to indicate the set of processing operations such as for storage and arithmetic operations. Normally, LET statement is used for such operations.



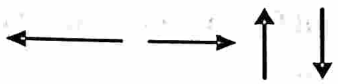
3. Input/Output Symbol: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are used for these operations.



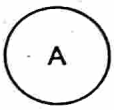
4. Decision Symbol: The diamond is used for indicating the step of decision-making and therefore known as decision box. It is used to indicate the process of logic in a program, which may be used for getting an answer to a question or may be for testing any condition. Computer based on the answer selects the further flow path. The decision box must have at least two exits, although a third may be added sometimes if desired.



5. Flow Lines: An arrow indicates the flow of operation. Every line in a flowchart must have an arrow on it.

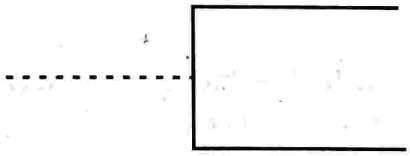


6. Connector: If the flowchart extends over more than one page, the different parts can be joined with a connector. A circle is used for this purpose. A letter or a digit is placed within the circle to indicate the link.



7. Predefined Process: A group of operations not detailed in the particular set of flowcharts.

8. Annotation or Comment Symbol: A broken line and bracket represent the annotation function—the addition of descriptive comments or notes for clarification of some statements.



Preparation of Flowcharts

General guidelines to prepare flowcharts are given below:

1. Ensure that the flowchart has a logical start and finish points.
2. Make the flowchart clear and easy to follow, so that it has a good visual impact.
3. Make comparison instruction simple, *i.e.* capable of yes/no (true/false) answers.
4. Try to construct flowchart from the top to bottom of a page and from the left to the right. If this convention is allowed, then a few arrows will be required.

5. Avoid crossed flow lines whenever possible.

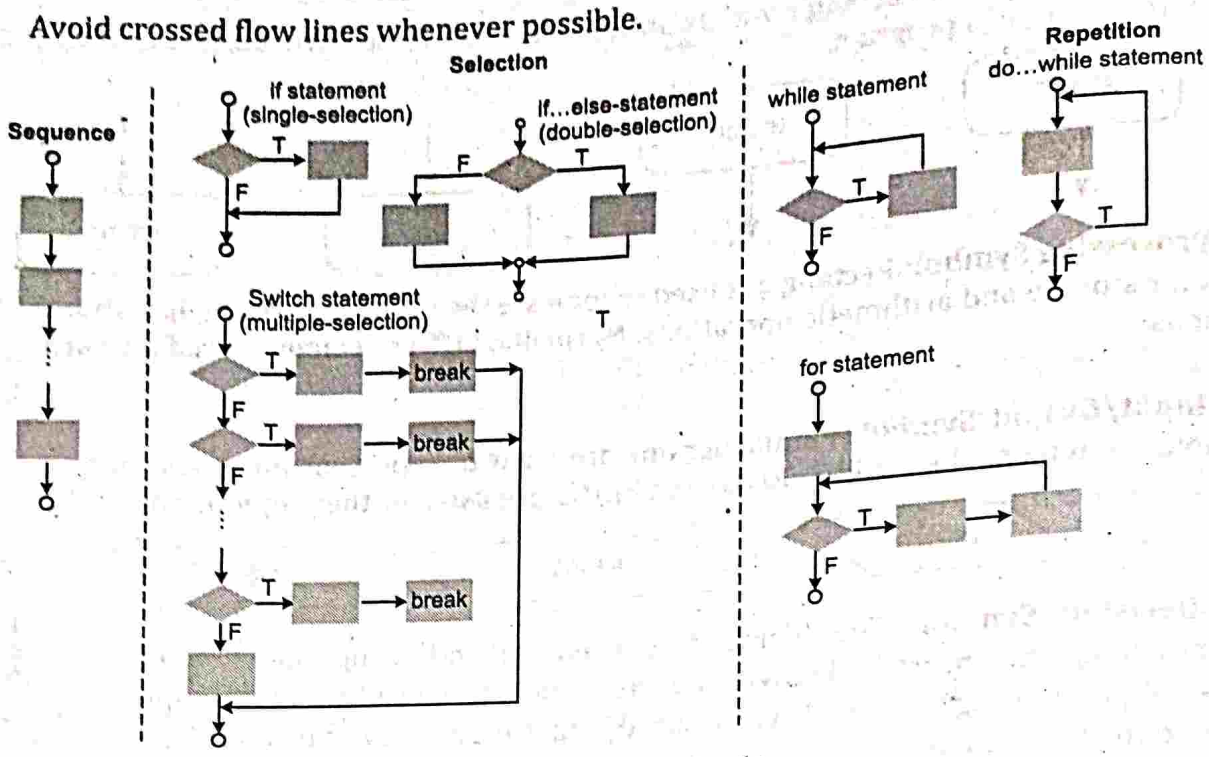


Fig. (a) Flowchart constructs.

Advantages of Using Flowcharts

The advantages of using flowcharts are as follows:

1. Flowcharts give a clear graphical representation of flow of program control.
2. Flowcharts are a good visual aid for communicating the logic of a system to all concerned.
3. They are language independent.
4. Current and proposed procedures may be understood more rapidly through the use of flowcharts. A programmer can chart a lengthy procedure more easily with the help of a flowchart than by describing it by means of written notes.
5. Flowcharts are useful in alterations of programs. The programmer can easily find the points in the flowcharts where changes are to be made.
6. Flowcharts are precise and do not include steps involved to embellish a program.

Limitations of Using Flowcharts

The various limitations of using flowcharts are as follows:

1. Alterations and Modifications: If alterations (or changes) are required in the program logic, the flowchart may require re-drawing completely.

2. Complex Logic: When the logic of the program is complex, the flowcharts quickly become complex and clumsy.

3. Standardisation: Program flowcharts are neither such a natural way of expressing procedures nor are they easily translated into programming language.

4. Reproduction: As the flowchart symbols cannot be typed, reproduction of flowcharts is often a problem.

5. Link between Conditions and Actions: Sometimes it becomes difficult to establish the linkage between various conditions and the related actions.

Q.7. What is big O (Oh) notation? Explain in detail.

Or **What is big Oh notation? Explain its use.** ***(2011)***

Or **What is big Oh notation? What are limitations and properties of big Oh nation? Explain in brief.** ***(2017)***

Ans.

Big O (Oh) Notation

In computer science, big O notation is used to classify algorithms by how they respond (*e.g.* in their processing time or working space requirements) to changes in input size. In mathematics, big O notation is used to describe the limiting behaviour of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions.

Knowledge Booster

Big O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used.

Big O notation characterises functions according to their growth rates. Different functions with the same growth rate may be represented using the same O notation. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols o, W, w, and Q, to describe other kinds of bounds on asymptotic growth rates.

Big O notation is also used in many other fields to provide similar estimates. The O notation for a function f(x) is derived by the following simplification rules:

If f(x) is a sum of several terms, the one with the largest growth rate is kept and all others omitted. If f(x) is a product of several factors, any constants (terms in the product that do not depend on x) are omitted.

Use of Big O notation

Big O notation has two main areas of application. In mathematics, it is commonly used to describe how closely a finite series approximates a given function, especially in the case of a truncated Taylor series or asymptotic expansion. In computer science, it is useful in the analysis of algorithms.

Informally, especially in computer science, the Big O notation often is permitted to be somewhat abused to describe an asymptotic tight bound where using Big Theta Q notation might be more factually appropriate in a given context. For example, when considering a function T(n) = 73n3 + 22n2 + 58, all of the following are generally acceptable, but tightnesses of bound (*i.e.*, bullets 2 and 3 below) are usually strongly preferred over laxness of bound (*i.e.*, number 1 below).

- $T(n) = O(n^{100})$, which is identical to $T(n) \square O(n^{100})$.
- $T(n) = O(n^3)$, which is identical to $T(n) \square O(n^3)$.
- $T(n) = \Theta(n^3)$, which is identical to $T(n) \square \Theta(n^3)$.

The equivalent English statements are respectively:

- T(n) grows asymptotically no faster than n^{100}.
- T(n) grows asymptotically no faster than n^3.
- T(n) grows asymptotically as fast as n^3.

So while all three statements are true, progressively more information is contained in each. In some fields, however, the Big O notation (number 2 in the lists above) would be used more commonly than the Big Theta notation (bullets number 3 in the lists above) because functions that grow more slowly are more desirable. For example, if T(n) represents the running time of a newly developed algorithm for input size n, the inventors and users of the algorithm might be more inclined to put an upper asymptotic bound on how long it will take to run without making an explicit statement about the lower asymptotic bound.

Limitations of Big Oh Notation: The limitations of big Oh notation are:

(a) Big Oh notation does not tell us which algorithm will be faster in any specific case. It only tells us that for sufficiently large input, one will be faster than the other.

(b) There may not be sufficient information to calculate the behaviour of the algorithm in an average case.

(c) Big Oh notation ignores the important constants sometimes. For instance, if a particular algorithm takes $O(n^3)$ time to run and another algorithm takes $O(100\ n^3)$ time to run, then both the algorithms would have equal time complexity.

(d) Big Oh notation tells us how the algorithm grows with the size of the problem and not the efficiency related to it.

Properties of Big Oh Notation: The properties of big Oh notation are:

(a) If the function f can be written as a finite sum of other functions, then the fastest growing one determines the order of f(n).

(b) If a function may be bounded by a polynomial in n, then as n tends to infinity, one may disregard lower-order terms of the polynomial.

(c) Any power of n may be ignored inside the logarithms. The set O (log n) is exactly the same as O (log n^c). The logarithms differ only by a constant factor and thus the big O notation ignores that.

(d) Changing units may or may not affect the order of the resulting algorithms. Changing units is equivalent to multiplying the appropriate variable by a constant whether it appears.

Section-A (Very Short Answer Questions)

Q. 1. Write a program to perform addition of two integers.

Ans.

```
/*Program to perform addition of two integers.*/
#include <stdio.h>
void main()
{ int number1, number2, result;
  printf("Enter the first number: ");
  scanf("%d", &number1);
  printf("Enter the second number: ");
  scanf("%d", &number2);
  result = number1 + number2;
  printf("Result of addition is: %d",result);
}
```

Output:

```
Enter the first number: 34
Enter the second number: 566
Result of addition is: 600
```

Q.2. Write a program to determine whether a number is positive or negative. *(2014)*

Ans.

```
/*Program to check whether a number is positive or negative.*/
#include <stdio.h>
void main()
{  double number;
   printf("Enter a number: ");
   scanf("%lf", &number);
   if(number < 0.0)
   printf("This number is negative");
  else
   printf("This number is positive");
 }
```

Output:

```
Enter a number: -3
This number is negative
```

UNIT 5

Syllabus

Simple Arithmetic Problems: Addition/Multiplication of integers, determining if a number is +ve/–ve/even/odd, maximum of 2 numbers, 3 numbers, sum of first n numbers, given n numbers, integer division, digit reversing, table generation for n, a^b , factorial, sine series, cosine series, ${}^nC_r$, pascal triangle, prime number factors of a number, other problems such as perfect number, GCD numbers,. etc. (write algorithms and draw flowchart), swapping.

Q.8. What do you understand by swapping? *(2016, 17)*

Or **Write a short note on swapping.** *(2015)*

Ans. Swapping is a simple memory/process management technique used by the Operating System (OS) to increase the utilisation of the processor by moving some blocked process from the main memory to the secondary memory (hard disk), thus forming a queue of temporarily suspended process and the execution continues with the newly arrived process. After performing the swapping process, the operating system has two options in selecting a process for execution:

(a) Operating system can admit newly created process (or)

(b) Operating system can activate suspended process from the swap memory.

Q.9. Perform swapping of two numbers with the help of variables. *(2013)*

Ans.

```
#include < stdio.h>
void main()
{    int X,Y, temp;
     printf( "Enter two no.")
     scanf ("% d % d,", &X, &Y);
     temp = X;
     X = Y;
     Y = temp;
     printf(" Numbers after swapping :%d %d", X,Y);
     getch();
}
```

Q. 2. Write a program to generate the table for a number.
Write a program in C to generate multiplication tables of 11 to 20. ***(2012)***

Ans.

```
//Program to generate the table for a number.
#include <stdio.h>
void main()
{  int a,n;
   printf("\nEnter a number : ");
   scanf("%d",&n);
   printf("Table of %d\n",n);
   printf("===========");
   for(a = 1 ;a <= 10 ;a++)
     printf("\n%d × %d = %d", n,a, n*a );
}
```

Output:

```
Enter a number: 12
Table of 12
==========
12 × 1 = 12
12 × 2 = 24
```

Q.5. Write a program to calculate the sum of first n numbers. *(2017)*
Or **Write a 'C' program to find the sum of the first n numbers.**

Ans.

```
//Program to calculate the sum of first n numbers.
#include <stdio.h>
void main()
{   int m, n, a, sum = 0;
    printf("Enter the value of n: ");
    scanf("%d",&n);
    for(a = 1; a <= n; a++)
      sum = sum + a;
    printf("Sum of first %d numbers is: %d", n, sum);
}
```

Output:

```
Enter the value of n: 10
Sum of first 10 numbers is: 55
```

Q.7. Write a program in C language to find a given number is prime. *(2015)*

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

```
#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.
```

Q.7. Write a recursive program to compute factorial of N numbers. *(2013)*

Ans.

```
/*Program to print factorial of N numbers.*/
#include <stdio.h>
long Fact(int num);
void main()
{     int n,num,a;
```

```
        long result;
        printf("How many numbers:");
        scanf("%d",&n);
        for(a = 1;a <= n;a++)
        {       printf("\nEnter a number:");
                scanf("%d",&num);
                result = Fact(num);
                printf("Its factorial is: %ld\n",result);
        }
}
long Fact(int num)
{       if(num == 0)
                return 1;
        else
                return num * Fact(num - 1);
}
```

Output:

```
How many numbers: 3
Enter a number: 6
Its factorial is: 720

Enter a number: 2
Its factorial is: 2

Enter a number: 8
Its factorial is: 40320
```

Q.9. How can you apply the recursion to solve ${}^nC_r$? *(2011)*

Ans.

```
/*Program to calculate nCr using recursion technique*/
#include <stdio.h>
#include <conio.h>
long Fact(int a);
void main()
{       int n,r;
        long fn, fr, fnr;
        float ncr;
        printf("Enter the value of n: ");
        scanf("%d",&n);
        printf("Enter the value of r: ");
        scanf("%d",&r);
        if(n - r < 0)
        printf("\nn must be equal or greater than r");
        else
        {       fn = Fact(n);
                fr = Fact(r);
                fnr = Fact(n - r);
                ncr = (float)fn / fr * fnr;
                printf("\nThe nCr is: %f",ncr);
        }
}
long Fact(int num)
{       if(num == 0)
                return 1;
        else
                return num * Fact(num - 1);
}
```

Output:

```
Enter the value of n: 5
Enter the value of r: 3
The nCr is: 40.000000
```

Q.14. Write a program to find out the GCD (Greatest Common Divisor) of three numbers. *(2013)*

Ans.

```
//Program to find the GCD of three numbers.
#include <stdio.h>
#include <conio.h>
#include <process.h>
int gcd(int m,int n)
{       int rem;
        while(n != 0)
        {       rem = m%n;
                m = n;
                n = rem;
        }
        return(m);
}
main()
{       int num1,num2,num3,gcd1,gcd2;
        clrscr();
        printf("Enter three positive integers");
```

```
        scanf("%d%d%d",&num1,&num2,&num3);
        if(num1 == 0 && num2 == 0 && num3 == 0)
        {       printf("\n Invalid number");
                exit(0);
        }       gcd1 = gcd(num1,num2);

                gcd2 = gcd(num3,gcd1);
                printf("\n GCD of %d %d %d is:
                %d\n",num1,num2,num3,gcd2);
}
```

Output:

```
Enter three positive integers 6
2
3
GCD of 6 2 3 is : 1
```

Q.15 Write a program in C to reverse the digits of a given number. *(2011)*

Ans.

```
// Program—To reverse the digits of a given number.
#include <stdio.h>
#include <conio.h>
void main()
{       int a, b, c;
        printf("Enter number:\n");
        scanf("%d", a);
        printf("Reversal of number:");
        do {
                b = a%10;
                printf ("%d", b);
                a = a/10;
                printf ("% d", b);
        } while (a > 0);
        getch();
}
```

Section-A (Very Short Answer Questions)

Q.1. Explain the concept of scope of variables. *(2014)*

Ans. The visibility or scope of a variable refers to those parts of a program, which will be able to access it directly. By using variables with the appropriate lifetime and visibility, we can write programs (or software) that use memory more efficiently, run faster, and are less prone to programming errors. 'C' language have four keywords for this purpose—`auto`, `static`, `register` and `extern`.

The scope of variables declared with the keywords auto, static or register is local, *i.e.* they are visible to the block (or function) in which the variable is declared. Whereas, the scope of external (or global) variables is global, *i.e.* they are visible to all the functions.

Q.2. What is the special character automatically added with the character array?

Ans. The null character is added automatically with the character array. It is denoted by '`\0`'. Its ASCII value is 0.

Example: `char st[] = "Rahul";`

The values stored in the character array `st` will be as shown below:

'R'	'a'	'h'	'u'	'l'	'\0'

Here, the null character ('\0') is not mentioned during the initialisation of the character array st; but, it is automatically added to `st`. Also the size of array `st` will automatically be 6 (counting of all the characters including the null character).

Q.3. Determine the output of the following program:

```
main()
{       int x,y,p,q;
        x = 5; y = 2;
        p = mul(x,y);
        q = mul(p, mul(x,z));
        printf("%d%d|n",p,q);
}
mul(a,l)
int a,l;
{
        return(a*l);
}
```

Ans. This program produces an error: "`Undefined symbol 'z' in function main()`".

UNIT 6

Syllabus

Functions: Basic types of function declaration and definition, functio call, types of function, Paramete passing, call by value, call b reference. scope of variable storage classes, recursion.

Section-B (Short Answer Questions)

Q.1. What do you understand by functions? Explain with its types. *(2015)*

Ans. Functions: Functions are a block of statements that perform a specific task. These statements take inputs, do some specific computation and then produces output. They process information which is passed to them from the calling portion of the program and return a single value.

There are two types of functions:

1. Standards Functions: The functions that perform most of the commonly used operations or calculations are standard functions. C language comprises of a library of standard functions. The functions that are inbuilt in the language compiler are library functions. *e.g.* `printf()` and `scanf()`.

2. User-defined Functions: These are the user modules that are deliberately created for a specific purpose. This creation of modules results in function. A user-defined function has to be developed by the user at the time of writing a program. It can later become a part of the C program library.

Q.2. Define local and global variables with example.

Or **Explain local and global variables.** *(2017)*

Ans. A quantity, which may vary during program execution, is called a variable. Variable names are those names given to locations in the memory of computer where different literals (or constant values) are stored. Two major types of variables are:

1. Local Variables: Those variables, which are declared inside a block or in the body of a function, are known as local variables. Local variables are only visible (*i.e.*, accessible) to the block (or function) in which they are declared. They cannot be accessed by other functions.

2. Global Variables: Those variables, which are declared outside all functions, are known as global variables. Global variables are visible (*i.e.*, accessible) to all the functions of the program. That means, the scope of global variables is global. In most cases, they are declared in the beginning of the program file.

To understand the local and global variables, consider the following program:

Program

```
/*Program to calculate the average of three numbers using global
and local variables*/
#include <stdio.h>
int n1, n2, n3;
float Average();
void main()
{       float a;
        n1 = 423;
        n2 = 34;
        n3 = 564;
        a = Average();
        printf("\nAverage is: %f",a);
}
float Average()
{       float avg;
        avg = (float)(n1 + n2 + n3)/3;
        return avg;
}
```

Output:

```
Average is: 340.333344
```

Output:

```
Average is: 340.333344
```

In the above program, `n1`, `n2` and `n3` are global variables and so they are accessible in bot `main()` and `Average()` functions. Whereas, `a` is a local variable of function `main()`; and so it i accessible only in `main()` and not in `Average()`. Similarly, `avg` is a local variable of functio `Average()` and so it is accessible only in `Average()` and not in `main()`.

Q.3. What are the header files? Explain their use in a 'C' program? Can you make out you header file? If yes, then how and if no, then why?
Or **What are header files in C? How do you use them? Also explain pre-processor directives in C with examples.** *(2012) (2013)*
Or **Enumerate importance of header files in C language.**

Ans. Header Files: The C compiler comes with a number of standard header files. These files contain declarations, definitions and macros that may be useful to a programmer in certain circumstances.

Uses of Header Files: When we use a library function in a 'C' program, we don't need to write the declaration or definition but, we only need to include that file in the beginning of the program using the `#include` preprocessor directive.

Making Our Own Header File: It is true that we can make our own header file. For it, create a new file and type some declarations and definitions in it and then save it with any extension (prefer .h). Now, to use this header file in the main file (that contains the `main()` function), include this file by giving its full name with `#include` preprocessor directive.

For example, if you create a header file with the name myheader.h, then include it in this way:

```
#include <myheader.h>
        or
#include "myheader.h"
```

Preprocessor Direction in C: Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. Commands used in preprocessor are called preprocessor directives and they begin with '#' symbol. Preprocessor directives used in C are #define, #undef, #include, #ifdef, #if, #endif, etc.

Example:

```
/*#if example */
#include <stdio.h>
#define MAX 100
int main(void)
{     #if Max > 99
      printf(''compiled for array greater than 99.\n'');
      #endif
      return 0;
}
```

Q.4. What is header files? Explain with its types. *(2016)*

Ans. Refer to Sec-B, Q.3.

Types of Header Files

There are mainly two types of header files:

1. **Library Header Files (or Predefined Header Files):** The header files that are written by compiler vendors are library header files. These are `stdio.h`, `conio.h`, `limits.h`, `float.h`, `string.h`, `stdlib.h`, etc.
2. **User Defined Header Files:** The header files that are written by programmers are user defined header files. For example;

```
#include <stdio.h>
#include <limits.h>
#include <math.h>
#ifndef alex_h
#define alex_h
void fibbo()
{     unsigned int i = 0, j = 1, sum = 0;
      int counter;
      printf("\n");
      for (counter = 1; counter <= 5000; counter++}
      {     if(sum >= INT_MAX)
                  break;
            else if(sum == 1836311903)
                  break;
            sum = i + j;
            i = j;
```

```
                    j = sum;
                    printf("\t% d", sum);
                }
        }
        long double factorial (unsigned int x)
        {       int counter;
                long double fact = 1;
                if (x == 0 || x == 1)
                return 1;
                else
                {       for(counter = x; counter >= 1; counter--)
                        fact = fact * counter;
                        return (long double) fact;
                }
        }
#endif.
```

Section-C *(Long Answer Questions)*

Q. 1. What is function? Explain with its examples. *(2016, 17)*

Or **What is a function in 'C'? How will you define and call a function? Explain with an example.**

Or **Describe the differences between actual and formal arguments.**

Or **What do you mean by modular approach? How functions in C support modular approaches? Also define function prototype, definition and calling with an example. Also list down its various advantages.** *(2013)*

Ans.

Modular Approach

In this approach, the problem to be solved is divided into the sub-problems or sub-modules. Each subproblem is solved separately, and then results are combined to get the desired solution. C is a structured programming language because to solve a large problem, C divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions.

Function: 'A function is a self-contained program structure that performs a specific task'.

Any 'C' program contains at least one function. If a program contains only one function, it must be `main()`. If there are more than one function, one of these must be `main()`, because program execution always begins with `main()`. There is no limit on the number of functions present in a 'C' program.

Some functions are already defined in 'C' library, they are called library functions. These library functions can be used in 'C' programs. Besides them, user can create some new functions, these functions are called user-defined functions.

There are three things related with a 'C' function—its definition, its declaration and its calling.

Definition: General syntax to define a function is:

Syntax:

```
returntype functionname([parameterlist])
{     //The body of method
}
```

The function definition consists of a line called the declarator, followed by the function body. The function body is composed of the statements that make up the function delimited by braces. The declarator of the function has three basic parts:

1. The type of the value that the method `returns (returntype)` could be a simple data type (such as `int`, `float`, etc.) as well as any user-defined data type. If the function does not return any value, it could even be void type. void is the keyword of 'C'.
2. The rules for naming a `function (functionname)` are same as for identifier. `Parentheses ()` are necessary after the function name.
3. An optional list of parameters, on which the function will `operate (parameterlist)`, must be enclosed in parentheses. This list contains the variable names with their types separated by commas. These are used to hold all the values the function will operate on. The syntax of parameter list is:

Syntax:

```
datatype variablename1, [datatype variablename2, ...]
```

Program 1

```
/*Program to calculate the volume using a function having three arguments and having a return type*/
#include <stdio.h>
//function declaration or prototype
double Volume(double w, double h,double d);
//definition of function main()
void main()
{  double width,height,depth,vol;
   width = 5.6;
   height = 3.4;
   depth = 2.3;
   //function calling
   vol = Volume(width,height,depth);
   printf("The volume is: %lf",vol);
}
//definition of function Volume()
double Volume(double w, double h, double d)
{  double v;
   v = w * h * d;
   return v;
}
```

Output:

```
The volume is: 43.792000
```

In the program 1, there are two functions defined in it. The first function is `main()`. Its return type is void because it does not return any value. The second function is: `Volume()`. It accepts three parameters of type double in local variables w, h and d. Then it has defined a local variable named v; calculate the volume and store in v. Then, it returns the value using the return statement. Since the value returned is of type double, the return type of method is also declared as double.

Calling the Function: A function is called (or referenced or invoked) by specifying its name within an expression. The call is terminated by a semicolon (;). Executing the call statement causes the function to execute, *i.e.* control is transferred to the function, the statements in the function body are executed and then control returns to the statement following the function call statement.

If we take the example of program 1, when `main()` executes, there is a calling statement for `Volume()` function:

```
vol = Volume(width,height,depth);
```

When this calling statement will execute, the 'C' run-time system transfers control to the first statement written inside the `Volume()` function. After the statements inside `Volume()` have executed, control is returned back to the calling routine (*i.e.*, `main()` function) and execution resumes with the line of code following the call.

Actual and Formal Arguments

A function can be passed on some values to work on. These values are called arguments (or parameters). When referencing (or invoking) a function, it can be followed by one or more arguments enclosed in parentheses and separated by commas. These arguments are called actual arguments. The actual arguments can be constants, variable names, subscripted variables or expressions.

The actual arguments will transfer information to formal arguments within the function definition. These arguments here are called formal arguments or dummy arguments. The formal arguments must be variables. The actual arguments, if any, must correspond in number, type and order with formal arguments.

If we take the example of program 1, there is a calling statement in function `main()`:

```
vol = Volume(width,height,depth);
```

In the above statement, width, height and depth are the actual arguments of the function `Volume()`. Consider the declarator part of the definition of function `Volume()`:

```
double Volume(double w, double h, double d)
```

In the above statement, `w`, `h` and `d` are formal arguments of the function `Volume()`.

The Function Declaration (Prototype)

A function prototype is a function declaration that specifies the data types of its arguments in the parameter list. The compiler uses the information in a function prototype to ensure that the corresponding function definition and all corresponding function declarations and calls within the scope of the prototype contain the correct number of arguments or parameters and that each argument or parameter is of the correct data type.

We can eliminate the function declaration if the function definition (the function itself) appears in the listing before the first call to the function.

So, when we specify the prototype of a function, we actually declare the function but not define it.

Function prototype declaration is a statement in which programmer describes the symbol name return type and arguments about a function.

Syntax: `<return type > < function name > () < arguments>;`

Its definition is the actual body/source code of the function and may contain one or more statements inside it.

Syntax:
```
<return type> <function name> () <arguments>
{     return;
}
```

Example:
```
int add(int a, int b)
{     return(a + b);
}
```

Advantages: Function prototype has the following advantages:

1. It helps the compiler in determining whether a function is called correctly or not.
2. A function must be defined before calling it. But prototyping allows a function to be called before defining it.

Q.2. How you can say C language supports modular approach? Why it is advantageous to use it? Explain the syntax of module declaration. Write a program using modular approach to find factorial of a number. *(2014)*

Ans. C language supports modular approach as it divides that problem into smaller modules called functions or procedures. Each of these procedures handles a particular responsibility.

It is advantageous to use this approach because of the following reasons:

1. Algorithm is more easily understood.
2. It allows library programs to be inserted.
3. Testing can be more thorough on each of the modules.
4. It causes fewer bug because each set of programming commands is shorter.
5. It saves time and means the finished program can be completed more quickly.

Syntax of Module Declaration: Module Attributes$_{opt}$ module Module fully Qualified Name;

Program using modular approach to find the factorial of a number:

```
#inclue <stdio.h>
int main (void)
{     int n;
      long factorial_r(int k);
      printf("Enter a positive number:")
      scanf("%i", &n);
      printf("Recursive:%i! is%1i\n",n,factorial_r(n));
      getch();
      return 0;
}
long factorial_r(int k)
{     if (k == 0) return1;
      else
      return k * factorial_r (k - 1);
}
```

Q.3. Explain types of user-defined functions with suitable example for each.
Or **Write a program which prints the sum of digits of an user-given number using a function.**
Or **Write a program to print the greater number between two integers using a function.**
Or **Write a program to find out whether the given number is even or odd.** ***(2016)***
Or **What is user-defined function? How can you return a value from a function? Explain with an example. Also discuss different methods to pass parameters to a function by giving suitable examples.** ***(2011, 12)***

Ans.

Categories of User-defined Functions

According to the arguments and return types, user-defined (and library) functions can be of four types:

I. Functions with Arguments and with Return Type

Those functions come in this category, which takes one or more arguments as well as return some value using the return statement. Function `DigitSum()` in program 1 comes in this category.

Program 1:

```
/* Program to print sum of digits of an user-given number using a function having one argument and with return type*/
#include <stdio.h>
int DigitSum(long num);
void main()
{     long num;
      int s;
      printf("\nEnter a number: ");
      scanf("%ld",&num);
      s = DigitSum(num);
      printf("Its digits' sum is: %d",s);
}
int DigitSum(long num)
{     int a,s = 0;
      while( num > 0 )
      {     a = num % 10;
            num = num / 10;
            s = s + a;
      }
      return s;
}
```

Output:

```
Enter a number: 4521
Its digits' sum is:12
```

II. Functions without Arguments and without Return Type

Those functions come in this category, which does not take any argument as well as return no value. Function `disp_msg()` of Program 2 comes in this category.

Program 2:

```
/*Program to display a simple message using a function having no
argument and no return type*/
#include <stdio.h>
void disp_msg();
void main()
{     disp_msg();
}
void disp_msg()
{     printf("\nHello World!");
}
```

Output:

```
Hello World!
```

III. Functions with Arguments and without Return Type

Those functions come in this category, which takes one or more arguments but return no value. Function `greater()` of Program 3 comes in this category.

Program 3:

```
/*Program to print the greater number among two integers using a
function having two arguments and no return type*/
#include < stdio.h>
void greater(int num1, int num2);
void main()
{     greater(34,62);
}
void greater(int num1, int num2)
{     if (num1 > num2)
      printf("\nGreater number is: %d",num1);
      else
      printf("\nGreater number is: %d",num2);
}
```

Output:

```
Greater number is: 62
```

IV. Functions without Arguments and with Return Type

Those functions come in this category, which take no arguments but return a value. Function `IsEven()` of Program 4 comes in this category.

Program 4:

```
/*Program to check whether integer is even or odd using a function having two arguments and no return type*/
#include <stdio.h>
int n;                              //global variable
int IsEven();                       //Prototype or Func. Declaration
void main()
{     int result;
      clrscr();
      printf("\nEnter a number: ");
      scanf("%d",&n);
      result = IsEven( );
      if(result == 1)
            printf("It is Even");
      else
            printf("It is Odd");
      getch();
}
int IsEven()
{     if( n%2 == 0 )
            return 1;
      else
            return 0;
}
```

Output:

```
Enter a number: 56
It is Even
```

Output:

```
Enter a number: 653
It is Odd
```

Q.4. Write short notes on `malloc()` and `free()` functions.

Or **Differentiate between `malloc()` and `calloc()` functions.**

Ans. Suppose, an array is declared of size 25 and at run-time user decides to handle less than 25 elements, then a lot of space of the array will be lost. If user decides to handle more than 25 elements, this extra space will not be available at run-time (or execution time).

This limitation of the array can be overcome if the memory is allocated at the execution time. There are some functions, such as `malloc()` and `calloc()`, which are used to allocate memory at the execution time. You must include the alloc.h header file to use these functions in the 'C' programs.

I. The malloc() Function

Knowledge Booster

'Malloc' returns a null pointer to indicate no memory is available or that some other error occurred which prevented memory being allocated.

The `malloc()` function gets a contiguous block of memory of specified size from the operating system and returns a pointer to the first location of this block. However, if contiguous memory block of specified size is not currently available, it returns a `NULL` pointer.

The simple form of using `malloc()` function is:

```
void *malloc(size);
```

Here, size is an integer value and denotes the number of bytes to be allocated. Notice that, this function returns a pointer to the type `void`. So, its programmer's duty to perform desired type conversion of the address returned by the function.

II. The calloc() Function

The `calloc()` function gets a contiguous block of memory of specified size from the operating system and returns a pointer to the first location of this block. However, if contiguous memory block of specified size is not currently available, it returns a `NULL` pointer.

The simple form of using `calloc()` function is:

```
void *calloc(nitems,size);
```

Here, `n items` is an integer and denotes the number of items for which memory is to be allocated. And size is an integer, which denotes the size of each item. Thus, the total memory allocated by `calloc()` is `n items*size` bytes.

Differences between malloc() and calloc():

1. After allocating the space, the `malloc()` does not initialise the allocated memory, whereas the `calloc()` initialise the allocated memory with 0.
2. The `malloc()` takes only one argument, whereas `calloc()` takes two arguments.

III. The free() Function

It will be programmer's duty to take care that the memory should be deallocated before the program terminates. This task is accomplished by using another library function `free()`. The `free()` function takes pointer to memory block as its argument and deallocates it.

Program:

```
/*Program to calculate squares of n numbers*/
/*Memory is allocated and deallcoated at run-time*/
#include <stdio.h>
#include <alloc.h>
#include <math.h>
void main()
{   double *arr,b;
    int n,a;
    printf("\nEnter, how many numbers: ");
    scanf("%d",&n);
    arr = (double*)malloc (n* sizeof(double) );
    printf("Enter any %d values",n);
    for(a = 0;a < n;a++)
    scanf("%lf", &arr[a] );
    printf("Squares are:\n");
    for(a = 0;a < n;a++)
    {     b = arr[a]*arr[a];
          printf("\n%.2f  %.2f",arr[a],b);
    }
    free(arr);
}
```

Output:

```
Enter, how many numbers: 3
Enter any 3 values
54
2
34
Squares are:
54.00        2916.00
2.00         4.00
34.00        1156.00
```

In the above program, suppose the value of variable n given by the user is 3. Since, size of `double` pe is 8, so, the memory allocated by `malloc()` is 24 (8 × 3). And the `free()` function deallocates is space at the end of the program.

Q.5. Write a short note on role of pointers.

Or **What are pointers? What are its advantages? Explain with example.**

Or **What is a pointer variable? What is the use of pointer variable?**

Or **Does C language support pointers? If yes, explain the syntax.** ***(2014)***

Ans.

Role of Pointers

Memory is allocated to the program variables by the operating system (such as Windows XP, NIX, etc.) during the compilation time. When the program is executing, the program can neither btain more of these variables nor deallocate (*i.e.*, free) the storage occupied by these variables to the perating system. To access a value of such variables, we simply use the name of these variables.

The 'C' language provides another approach, which enables the program to acquire as much as torage required to store more values and at the same time allow the program to deallocate the torage, which is no more required, by the program. This is accomplished through the use of special ariables called pointer variables (or simply pointers). Thus, C language supports pointers.

"A variable that holds an address is called a pointer variable."

ypes of Pointers

Based on the fact that whether a pointer variable holds the address of an integer, a float, or a haracter variable, the pointers are classified as integer pointers, real pointers, character pointers nd so on.

Zero pointer or null pointer is normally used to indicate that the pointer does not contain an ctive entry. They are normally initialised with NULL. NULL is a pre-defined symbolic constant in `TDIO.H`.

eclaring a Pointer Variable

Like other variables, the pointer variables are to be declared to tell the 'C' compiler that to which ind of values (or constants) these special variables will be pointing.

Syntax: `type *pointer_variable_name;`

where `type` is a data type and indicates that the pointer will point to the variables of that specific ata type and character * indicates that variable is a pointer variable.

Example: `int *ptr;`

The above type declaration statement, declares a pointer variable of `int` type, *i.e.* pointer ariable that will hold the address of storage location where some integer value is stored.

nitialising a Pointer Variable

If a pointer is not properly initialised (*i.e.*, it contains garbage value), it may point to any location n memory including those locations where operating system is running and our system may hang-up (if used). Such un-initialised pointers are sometimes referred to as damaging pointers.

So, it is necessary to initialise the pointer variable before using it. Consider the following example:

Example:

```
int var = 1234;
int *ptr;
ptr = &var;
```

In the above statements, `var` is a simple variable, which holds an integer value `1234`. `ptr` is a pointer variable that will point to an integer. `&` is an 'address of' operator, which finds the address of variable `var`; and then the assignment operator(=) stores this address in the pointer `ptr` (Refer the figure below).

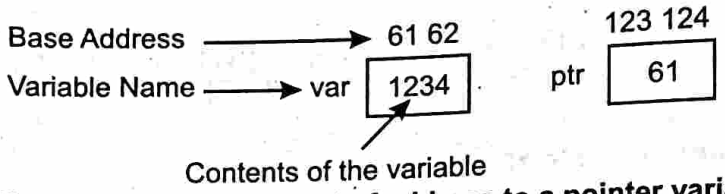


Fig. Illustration of assignment of address to a pointer variable.

It is clear from the figure above that the base address (*i.e.*, address of first byte of the variable) of variable `var` is (suppose) 61. After assigning this address to pointer `ptr`, the contents of `ptr` becomes 61, *i.e.* pointer `ptr` is now pointing to integer variable var.

Also notice that `ptr` also has its own address, since it is a variable. In this case, `ptr` will consume two bytes in memory to hold the address.

Accessing Values using Pointers

Indirection operator (*) is used to access the value pointed by a pointer variable. It is also known as value at address operator or direction of operator. To understand its use, consider the example below:

Example:

```
int *iptr, a = 531;        iptr = &a;
 :
printf("Value of var. a = %d\n", a);
printf("Value of var. pointed to by iptr = %d\n", *iptr);
printf("Address of var. a = %d\n", iptr);
 :
```

Knowledge Booster

A pointer references a location in memory and obtaining the value stored at that location is known as dereferencing the pointer.

In the above example, the declaration statement tells the compiler that `iptr` is a pointer variable that will point to an integer value, `a` is an integer variable and it is initialised with value `531`. The address of `a` is assigned to the pointer `iptr`. And suppose that during the execution of the above segment, a memory location with address `1597` is set aside for variable `a`. The output of the above statement will look like:

```
Value of var. a = 531
Value of var. pointed to by iptr = 531
Address of var. a = 1597
```

Advantages or Uses of Using Pointers

Some of the advantages offered by the use of pointers are:

1. The pointer notation compiles into faster and more efficient code.
2. Pointers are used to reduce the length and complexity of programs.
3. Pointers are used to communicate with operating system about memory. That is, it allows dynamic allocation and deallocation of memory segments.

4. Using pointers, we can return more than one values from a function.
5. We can create functions using pointers, which can modify their actual arguments.
6. Using pointers we can pass arrays and strings more efficiently as arguments to a function.
7. Pointers are used to create complex data structures, such as linked lists, binary trees and graphs, where one data structure element must contain references to the other data structures of same type.
8. Like simple variables, we can add (and subtract) a number to/from a pointer. We can also subtract one pointer variable from another pointer variable.

imitations in Using Pointers

The following arithmetic operations on pointers are not permitted:

1. We cannot add two pointer variables.
2. We cannot multiply a pointer variable by a number.
3. We cannot divide a pointer variable by a number.

Q.6. Show the relation between a pointer and an array.
Or **How can you pass one-dimensional array to a function as an argument? Explain.**

Ans.

Relation between Pointer and an Array

Consider the following declaration statement:

```
float a[9] = { 0,1,2,3,4,5,6,7,8 };
```

This statement set aside a contiguous block of nine memory locations for the array named a. lere the array name a (without index) denotes the address of the first element of the array, *i.e.* base ddress of the array. The memory map of the array a is shown in figure below:

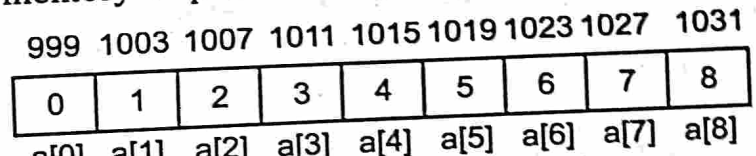


Fig. Memory map of one-dimensional array a with a base address 999.

According to the figure, the base address of a is 999 (there may be a different base address in ther computers). By incrementing this base address, the other elements of the array can also be ccessed. For example, using pointer notation, the elements of the array a can be accessed as

```
*(a + 0), *(a + 1), *(a + 2), ..., *(a + 8)
```

In comparison to subscript notation, accessing array elements using pointers is always faster.

Passing One-Dimensional Array to a Function as an Argument

We can pass an entire array as an argument to a function. In mostly cases, the name of the array and total number of elements are passed as actual arguments.

For example, if you call the function named display(), then the statement

```
display( &num[0],6);
```

or

```
display(num,6);
```

will pass the array num by passing address of first element and total number of elements. In the function definition, it will be accepted as:

```
display( int *arr, int size)
{
      :
}
```

or

```
display( int arr[], int size)
{
      :
}
```

Notice that the changes made to the array will actually be made in the original array that means array is passed by reference.

Q.7. What is the differences between call by value and call by reference?

Or **Explain all the approaches to passing arguments to a function in 'C'.**

Or **Distinguish between 'call by value' and 'call by reference' methods of parameter passing by giving suitable examples.** *(2011)*

Or **Explain call by value and call by reference. Which method is more memory efficient and why?** *(2013)*

Or **Distinguish call by value and call by reference.** *(2014)*

Or **What do you understand by call by value?** *(2016)*

Ans. **Approaches of Passing Arguments to a Function**

There are two approaches to passing arguments to a function. These are: 1. Call by value 2. Call by reference.

I. Call by Value

This approach is also popularly known as passing arguments by value. In this approach, the values of the actual arguments are passed to the function during function call. When control is transferred to the called function, the values of the actual arguments are copied to the corresponding formal arguments and then, the body of the function is executed. If the called function is supposed to return a value, it is returned with the help of return statement.

To understand the concept of call by value, consider the program 1.

Program 1:

```
// Example of Passing arguments to a function by value
#include <stdio.h>
#include <conio.h>
#include <math.h>
void Line(char, int);
void main()
{     int b = 8;
      clrscr();
      Line('&', pow(8,2) );
      printf("\n\t\t\t\t    W E L C O M E");
      Line('$',68);
      printf("\n\t\t\t\t T O   K H A T A U L I");
      Line('#',60);
      getch();
}
void Line(char ch, int n)
```

```
{       int a;
        printf("\n");
        for (a = 1;a < = n;a++)
              printf("%c",ch);
}
```

Output:

```
& & & & ..................................... & & & &
                  W E L C O M E
$ $ $ ..................................... $ $ $ $
                T O   K H A T A U L I
# # ..................................... # #
```

In the above program, the actual arguments are values, such as `&, 64, $, 68,` etc. These values are copied to the formal arguments `ch` and `n`.

II. Call by Reference

This approach is also popularly known as passing arguments by reference. In this approach, the addresses of the actual arguments are passed to the function during function call. When control is transferred to the called function, the addresses of the actual arguments are copied to the corresponding formal arguments and then, the body of the called function is executed.

In this case, the formal arguments must be declared as pointers to types that match the data types of the actual arguments. This approach is of practical importance while passing arrays and strings to function and also for passing back more than one value to the calling function.

To understand the concept of call by reference, consider the program 2.

Program 2:

```
#include <stdio.h>
void Swap(int *m, int *n);      //Prototype
void main()
{       int a = 5, b = 7;
        Swap(&a , &b);
        puts("After swapping:");
        printf("a is %d, b is %d",a,b);
}
void Swap(int *m, int *n)
{       int p;
        p = *m;
        *m = *n;
        *n = p;
}
```

Output:

```
After swapping:
a is 7, b is 5
```

In the program 2, the addresses (*i.e.*, references) of variables `a` and `b` are passed to the function `Swap()`. These addresses are copied to formal arguments m and n, which are declared as pointers to hold the addresses.

Both kinds of parameter calls may be used in subprogram arguments. The differences between call by value and call by reference are summarised below:

S.No.	Basis of difference	Call by value	Call by reference
1.	Actual parameter	The actual parameter may be constant, variable, function call, or a combination of these.	The actual parameter must be variable.
2.	Formal parameter	The formal parameter assumes only the value of the actual parameter.	The formal parameter assumes the address of the actual parameter.
3.	Value of actual parameter	With the help of formal parameter, value of actual parameter cannot be changed.	With the help of formal parameter, value of actual parameter can be changed.
4.	No. of values returned	Only a single value can be returned from a function, using a return statement.	More than one value can be returned (indirectly) from a function, without using the return statement.

Q.8. Describe the differences between automatic and static variables. *(2011)*

Or **What is 'static' storage class?** *(2012)*

Or **What are storage classes in C? Explain each with an example.** *(2013)*

Or **Explain the concept of storage classes.** *(2015)*

Or **What do you understand by storage classes? Explain.** *(2017)*

Or **Explain storage class.**

Ans.

Storage Classes

Every variable in 'C' holds a characteristic called its storage class. The storage class defines mainly two characteristics of the variable:

1. Lifetime: The lifetime of a variable is the length of time it retains a particular value.

2. Visibility or Scope: The visibility or scope of a variable refers to those parts of a program, which will be able to access it directly.

By using variables with the appropriate lifetime and visibility, we can write programs (or software) that use memory more efficiently, run faster and are less prone to programming errors. 'C' has four keywords for this purpose— `auto`, `static`, `register` and `extern`. Four storage classes are:

I. Automatic Storage Class

The variables declared with `auto` storage class are known as automatic (or Local or Internal) variables, which is also the storage class by default. They have the following features:

1. Stored in memory.
2. If not initialised in the declaration statement, their initial value is unpredictable, which is often called garbage value.
3. Local (visible) to the block (or function) in which the variable is declared.
4. It retains its value till the control remains in the block (or function) in which the variable is declared. As the execution of the block (or function) terminates, it is cleared/destroyed and their memory space gets free.

II. Static Storage Class

The variables declared with `static` storage class (known as static variables) have the following features:

1. Stored in memory.
2. If not initialised in the declaration statement, their initial value is zero.
3. Local (visible) to the block (or function) in which the variable is declared.
4. It retains its value between different function calls, *i.e.* when for the first time a function is called, a static variable is created with initial value zero and in subsequent calls it retains its present value.

I. Register Storage Class

The compiler, if possible, assigns a `register` variable to one of the processor's register instead f storing it in the memory. Value stored in register can be accessed much faster than the value stored n memory. Here the address of operator (&) cannot be applied to register variables. The register ariables have the following features:

1. Stored in processor registers, if a register is available. If no register is available, the variable is stored in memory and works as if its storage class is `auto`. As soon as register becomes available, the storage class of that variable is converted to register type.
2. If not initialised in the declaration statement, their initial value is unpredictable, which is often called garbage value.
3. Local (visible) to the block (or function) in which the variable is declared.
4. It retains its value till the control remains in the block (or function) in which the variable is declared. As the execution of the block (or function) terminates it is cleared/destroyed and their memory space gets free.

V. External Storage Class

The scope of external variables is global. External variables are declared outside all functions, *i.e.* n the beginning of the program file. As all the function will be defined after these variable declarations, so these variables are visible to all functions in a program file.

The external (or global) variables have the following features:

1. Stored in memory.
2. If not initialised in the declarations statement, their initial value is zero.
3. Global, *i.e.* visible to all functions.
4. Retains its values till the program execution terminates.

Suppose we have defined two user-defined functions `fun1()` and `main()` and we have defined a global variable after the definition of `fun1()`. To use this global variable in the definition of `fun1()` without error, we have to declare that global variable with `extern` keyword in the `fun1()`.

Syntax: `extern type variable_name;`

Also note that, extern declaration does not allocate storage space for variables.

Consider the following program, which illustrates the difference between variables of automatic and static storage class:

Program:

```
    /*Program illustrating the use of automatic and static
variables*/
    #include <stdio.h>
    void fun();
    void main()
    {       fun();
            fun();
            fun();
    }
    void fun()
```

```
{       auto int a = 0;
        static int b;
        a++;
        printf("\nAutomatic variable: %d",a);
        b++;
        printf("\nStatic Variable: %d",b);
}
```

Output:

```
Automatic Variable: 1
Static Variable: 1
Automatic Variable: 1
Static Variable: 2
Automatic Variable: 1
Static Variable: 3
```

Q.9. Write a short note on recursion. *(2011, 12)*

Or **What is the concept of recursion? Apply recursion to develop the program in 'C' for calculating the factorial of a number.**

Or **What are the recursion functions and how are they defined? Define any two recursive functions.**

Or **Write a program to find the factorial of a given positive number.**

Or **What is recursion? Explain with its suitable example.** *(2015)*

Or **Write a program in C for recursion.** *(2016)*

Or **Explain recursion with suitable example.** *(2017)*

Ans.

Recursion

In mathematics and computer science, recursion means self-reference. Recursion is a repetitive process in which a function calls itself again and again till a termination condition is true (or false). Using this technique, we can solve those problems, which are defined in terms of themselves. Consider the following examples:

Examples:

1. $5! = 5 \times 4!$
2. $(4)^3 = 4 \times 4^2$

According to the first example, factorial of 5 can be finally calculated after calculating the factorial of 4. Similarly, $(4)^3$ can be calculated after calculating the $(4)^2$. That means, these operations are defined in terms of themselves.

For using recursion technique in programming, the programming language must support this technique. 'C' supports recursion technique.

Knowledge Booster

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until one get to a small enough problem that it can be solved trivially.

A function containing either a call statement to itself or a call statement to a second function that eventually may result in a call statement back to the original function, is called a recursive function. In case of recursion, the calling function and the called function are same. So that the program will not continue to run indefinitely.

A recursive function must have the following two properties. A recursive function with these two operties is said to be well-defined.

1. There must be certain criteria, called base criteria, for which the function does not call itself.
2. Each time the function does call itself, it must be closer to the base criteria.

To understand the concept of recursion technique, let us create a program which calculates the torial of a number using a recursive function. The recursive definition of factorial can be written as:

$$N = \begin{cases} 1, & N = 0 \\ N * (N-1)! & N > 0 \end{cases}$$

Program:

```
/*Program to calculate factorial of a number using recursion technique*/
#include <stdio.h>
#include <conio.h>
long Fact(int a);
void main()
{  long n;
   int a;
   printf("\nEnter a number: ");
   scanf("%d",&a);
   n = Fact(a);
   printf("\nFactorial of %d is: % ld",a,n);
   }
   long Fact(int num)
   {     if(num == 1)
               return 1;
         else
               return num * Fact(num - 1);

   }
```

Output:

```
Enter a number: 5
Factorial of 5 is: 120
```

Let us see a recursive function Power() to calculate the power of a number for a given exponent's value.

```
long Power(int x, int n)
{     if(n == 0)
            return 1L;
      else
            return n * Power(x,n - 1);

}
```

For the above function, if actual arguments are 2 and 5 for the formal arguments x and n, then the returned value will be 32.