1. **(i) Describe the structure of a C program with an example program. (16)**

Documentation section
Preprocessor section
Definition section
Global declaration section

Void Main()
{
Declaration part;
Executable part;
}

Sub program section
{
Body of the sub program;
}

Documentation section:

- It contains a set of comment lines used to specify the name of program the author and other details etc.,

Comments:

- Comments are very helpful in identifying the program features and underlying logic of the program.
- The single line comments using "\\".
- The lines begins with '/*' and ending with '*/' are known as comment lines. These are not executable, the compiler is ignored anything in between /* and */.

Preprocessor section:

- It is used to link system library files for defining the macros and for defining the conditional inclusion.

  Eg: #include<stdio.h>, #define A 10, #if def , #endif....etc.

Global declaration section:

- The variables that are used in more than one function throughout the program are called global variables and declared outside of all the function i.e., before main().
- Every 'C' program must have one main() function, which specify the starting of 'C' program. It contains the following two parts.

Various elements of 'C' program:

Comment: /* program for temperature conversion */

Header file: #include<stdio.h>

Preprocessor directories: #define con 1.8 //Constant

Reserved words: int main ()

{

Variable type: float c,f //Standard identifiers

Control string: printf("enter the Celsius values.....");

Special character: scanf("%f", &c);

Execution character: f = (con * c)+32;

printf("\n Fahrenheit value of the given %f Celsius value is %f", c,f);

Reserved word:return(0);

}

Declaration part:

- This part is used to declare all the variables that are used in the executable part of the program and these are called local variables.

Executable part:

- It contains atleast one valid 'C' statement.
- The execution of a program begins with opening brace '{'and ends with closing brace '}'.
- The closing brace of the main function is the logical end of the program.
- All the statements in the program ends with a semicolon(;) except conditional and control statements.

---

2. (i) Describe in detail about different data types in 'C' with suitable examples. (8)

Data types:

- Data type is the type of the data, that are going to access within the program.
- Each data type may have predefined memory requirement and storage representation.
- Data type or just type determines the possible values that an identifier can have and the valid operations that can be applied on it. Data types are broadly classified

Primary	**User defined**	**Derived**
char	Structures	Arrays
int	Union	Pointers
float	Enumeration	Function
Double		
void		

1.Basic data types (primitive data types)

a) Character (char)
b) Integer (int)
c) Single-precision floating point (float)
d) Double-precision floating point (double)
e) No value available (void)

2.Derived data types:

These data types are derived from the basic data types.

a) Array type e.g. char[ ], int[ ], etc.
b) Pointer type e.g. char *, int *, etc.
c) Function type e.g. int (int, int), float (int), etc

1. User-defined data types:

These data types can be created by using:

a) Structure
b) Union
c) Enumeration

2.) (ii) Define constants. Explain the various types of constants in C. (8)

A constant is an entity whose value remains the same throughout the execution of a program. It cannot be placed on the left side of the assignment operator because it does not have a modifiable L-value. It can only be placed on the right side of the assignment operator.

1. **Literal Constant**:
2. **Qualified Constant**:
3. **Symbolic Constant**:

1. **Literal Constant**:

i. Integer Literal Constant:

i. Integer literal constants are integer values like –1, 2, 8, etc. The rules for
ii. writing integer literal constants are:
iii. a. An integer literal constant must have at least one digit and should not have any decimal point.
iv. b. It can be either positive or negative. No special characters are allowed within an integer literal.
v. c. If an integer literal constant starts with 0 (zero), then it is assumed to be in an octal number
vi. system, e.g. 023 is a valid octal integer literal constant [i.e. $(23)_8 = (19)_{10}$].
vii. d. If an integer literal constant starts with 0x or 0X, it is assumed to be in a hexadecimal number
viii. system, e.g. 0x23 or 0X23 is a valid hexadecimal integer literal constant [i.e. $(23)_{16}=(35)_{10}$].
ix. e. The size of the integer literal constant can be modified by using a length modifier.
x. – If the integer literal constant is terminated with l or L then it is assumed to be long (e.g. 23l).
xi. – If it is terminated with u or U, then it is assumed to be an unsigned integer (e.g. 23u).

ii. Floating Point Literal Constant:

Floating point literal constants are values like –23.1, 12.8, –1.8e12, etc. These constants can be written in a fractional form or in an exponential form. The rules for writing floating point literal constants in a fractional form are:

a. A fractional floating point constant must have at least one digit and should have a decimal point.
b. It can be either positive or negative. No special characters are allowed within a floating point literal constant.
c. A floating point literal by default is assumed to be of type double, e.g. the type of 23.45 is double.
d. The size of the floating point literal constant can be modified by using the length modifier f or F, i.e. if 23.45 is written as 23.45f or 23.45F, then it is considered to be of type float.

The rules for writing floating point literal constants in an exponential form are:

a. A floating point literal constant in an exponential form has two parts: the mantissa and the exponent. Both parts are separated by e or E.
b. The mantissa can be either positive or negative and should have at least one digit. The mantissa part can have a decimal point but it is not mandatory.
c. The exponent part must have at least one digit and can be either positive or negative. The exponent part cannot have a decimal point.
d. No special characters are allowed within the mantissa part and the exponent part. Example: 2e10 (i.e. equivalent to $2 \times 10_{10}$)

iii. Character Literal Constant:

A character literal constant can have one or at most two characters Enclosed within single quotes, e.g. 'A', 'a', '\n', etc. Character literal constants are classified as:

a. Printable Character Literal Constant: All characters of source character set except quotation mark, backslash and new line character when enclosed within single quotes form a printable character literal constant.

Examples: 'A', '#', '6', etc.

b. Non-printable Character Literal Constant: These constants are represented with the help of escape sequences. An escape sequence consists of a backslash (i.e. \) followed by a character and both enclosed within single quotes. An escape sequence is treated as a single character and can be used in a string like any other printable character. A list of escape sequences are given below:

iv. String Literal Constant:

A string literal constant consists of a sequence of characters enclosed within double quotes.

Each string constant is implicitly terminated by a null character (i.e. '\0').

Hence, the number of bytes occupied by a string constant is one more than the number of characters present in the string. However, the terminating null character is not counted while determining the length of a string.

For example,

the length of the string "ABC" is 3 although it occupies 4 bytes of memory.

2.Qualified Constant:

Qualified constants are created by using const qualifier. Since qualified constants are placed in the memory, they have L-value. However, as it is not possible to modify them, they do not have a modifiable L-value.

Example: const int a=10.

3.Symbolic Constant:

Symbolic constants are created with the help of the define pre-processor directive. Each symbolic constant is replaced by its actual value during the pre-processing stage.

For Example, #define PI 3.1429 defines PI as a symbolic constant with value 3.1429.

3).Explain the different types of operators available in C with example.(16)
(May 2015) 4
Using Operators in "C"

The operators in C are classified into two categories.

a. **Classification Based on the Number of Operands**
b. **Classification Based on Role of Operator**

1. Classification Based on the Number of Operands

Based upon the number of operands on which an operator operates, the operators are classified as:

i) Unary operators :

A unary operator operates on only one operand. The operand can be present towards the right (e.g. –a) or towards the left of the unary operator (e.g. a++).

Examples: – (unary minus), & (address-of-operator), sizeof operator, (logical negation), ~ (bitwise negation), ++ (increment), -- (decrement), etc.

ii) Binary operators :

A binary operator operates on two operands. It requires an operand towards its left and right.

Examples: * (multiplication operator), / (division operator), << (left shift operator), == (equality operator), && (logical AND), & (bitwise AND), etc.

iii) Ternary operators :

A ternary operator operates on three operands. Conditional operator (?) is the only ternary operator available in C.

Classification Based on Role of Operator

Based upon the number of operands on which an operator operates, the operators are classified as:

i) Arithmetic operators: Arithmetic operations like addition, subtraction, multiplication, division,etc. can be performed by using arithmetic operators as shown below

S.No.	Operator	Name of Operator	Category	Type	Precedence Among Arithmetic Class	Associativity
1.	+ – ++ --	Unary plus Unary minus Increment Decrement	Unary Operators	Unary	Level-I (Highest)	R ⇗ L (Right-to-Left)
	*	Multiplication	Multiplicative	Binary	Level-II	L ⇗ R

2.	/ %	Division Modulus	Operators		(Intermediate)	(Left-to-Right)
3.	+ –	Addition Subtraction	Additive Operators	Binary	Level-III (Lowest)	L ⇗ R

While evaluating arithmetic operations:

a. The parenthesized sub-expressions are evaluated first. If the parentheses are nested, the innermost sub-expression is evaluated first.

b. The precedence rules are applied to determine the order of application of operators in evaluating sub-expressions. The associativity rule is applied when two or more operators of the same precedence appear in the sub-expression.

c. If the operands of a binary arithmetic operator are of different but compatible types, C automatically applies arithmetic-type conversion or implicit-type conversion or type promotion. If operands are of different types, the lower type (i.e. smaller in size) should be converted to a higher type (i.e. bigger in size) so that there is no loss in value or precision.

Binary arithmetic operators can be used in one of the following three different modes:

a. Integer mode : If both the operands of a binary arithmetic operator are of integer type, the mode of operations is said to be integer mode and the result will be of integer type.
Example: The result of 4 / 3 will be 1 instead of 1.3333.

b. Floating point mode : If both the operands of a binary arithmetic operator are of floating point type, the mode of operation is said to be floating point type and the result will be of floating point type.
Example: 4.0 / 3.0 = 1.333333.

c. Mixed mode : If one of the operands of a binary arithmetic operator is of integer type and another operand is of floating point type, the mode of operation is said to be mixed mode. The operand of integer type is promoted to floating point type and the result will be of floating point type.

Example: 4/3.0=1.333333.

#include<stdio.h>
main()
{
int a;
a = 2 * 3.25 + ((3 + 6) / 2);
printf("Result is %d", a);
}
	The evaluation of the expression given in the program is as follows:
	a = 2 * 3.25 + ((3 + 6) / 2);
	= 2 * 3.25 + (9 / 2);
	= 2 * 3.25 + 4;
	= 6.50 + 4;
	= 10;

Output: Result is 10	The variable a is declared as int, and hence the value (i.e. 10.50) is automatically converted to an integer type and assigned to a. Since a higher type is converted to a lower type, it is said that the higher type is demoted to the lower type and this conversion is called demotion or truncation.

Increment / Decrement Operator:

It has two forms:

– **Pre-increment / Pre-decrement operator**: The operator appears towards the left side of its operand (e.g. ++a or --a).

– **Post-increment / Post-decrement operator:** The operator appears towards the right side of its operand (e.g. a++ or a--).

Increment / decrement operator is a token (i.e. one unit). There should be no white-space

character between two '+' symbols or two '–' symbols. If white space is placed between two '+' symbols or two '–' symbols, they become two unary plus (+) or two unary minus (–) operators.

The increment / decrement operator can only be applied to an operand that has a modifiable l-value. The value of ++a or a++ is equivalent to a = a + 1. Similarly the value of --a or a-- is equivalent to a = a – 1.

Difference between pre-increment / pre-decrement and post-increment / post-decrement operator:

– In case of the pre-increment / pre-decrement operator, first the value of its operand is incremented / decremented and then it is used for the evaluation of expression.

– In case of the post-increment / post-decrement operator, the value of operand is used first for the evaluation of the expression and after its use the value of operand is incremented / decremented.

#include<stdio.h>
main()
{
 	
int a = 2, b = 2, c, d;
 	
c = ++a;
 	
d = b++;
printf("a=%d, b=%d, c=%d, d=%d", a, b, c, d);
 	
}
	Output:
	a=3, b=3, c=3, d=2
	
	Reasons:
	
	• The value of a is incremented and then it is assigned to c.
	
	• The value of b is assigned to d before it is incremented.

Modulus Operator: The modulus operator (%) is used to find the remainder. The operands of modulus operator must be of integer type. The sign of the result of evaluation of modulus operator depends only on the sign of the numerator.

ii) **Relational operators:** An expression that involves a relational operator forms a condition. The result of evaluation of a relational expression (i.e. involving relational operators) is a boolean constant,

i.e. 0 (false) or 1 (true). The relational operators shown below are used to compare two operands.

S.No.	Operator	Name of Operator	Category	Type	Precedence Among Relational Class	Associativity
1.	< ? <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-I	L ↱ R
2.	== !=	Equal to Not equal to	Equality operators	Binary	Level-II	L ↱ R

Example:
#include<stdio.h>
main()
			Output:
			The value of a is 1.
			Reason:			

{
int n;
 	
a = 2 < 3 != 2;
 	
printf("The value of a is %d", a);
}
	• The expression a = 2 < 3 != 2 is interpreted as a = (2 < 3) != 2. The sub-expression 2 < 3 is true (i.e. 1). 1 != 2 is true (i.e. 1).
	
	So, 1 is assigned to a.

iii) Logical operators: The logical operators listed below are used to logically relate the sub-expressions.

S.No.	Operator	Name of Operator	Category	Type	Precedence Among Logical Class	Associativity
1.	!	Logical NOT	Unary	Unary	Level-I	R ⇗ L
2.	&&	Logical AND	Logical operator	Binary	Level-II	L ⇗ R
3.	||	Logical OR	Logical operator	Binary	Level-III	L ⇗ R

Logical operators consider operand as an entity (i.e. a unit) and they operate according to the truth table given below:

AND Operation		
Operand1	Operand2	Operand3
False	False	False
False	True	False
True	False	False
True	True	True

OR Operation		
Operand1	Operand2	Operand3
False	False	True
False	True	True
True	False	True
True	True	True

NOT operation	
Operand	Result
False	True
True	False

If an operand of a logical operator is a non-zero value, the operand is considered as true. If the operand is zero, it is considered as false. Each of the logical operators yields 1 if the specified relation evaluates to true and 0 if it evaluates to false. The result is of type int.

In an expression E1 && E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to 0 (i.e. false), E2 will not be evaluated and the result of the overall expression will be 0 (i.e. false). If E1 evaluates to a non-zero value (i.e. true) then E2 will be evaluated to determine the truth value of the overall expression.

In an expression E1 || E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to a non-zero value (i.e. true), E2 will not be evaluated and the result of the overall expression will be 1 (i.e. true). If E1 evaluates to 0 (i.e. false), then E2 will be evaluated to determine the truth value of the overall expression.

iv) Bitwise operators: The C language provides six operators for bit manipulation and operates on the individual bits of the operands. These operators can only be applied on operands of type char, short, int, long, whether signed or unsigned.

S.No.	Operator	Name of Operator	Category	Type	Precedence Among Bitwise Class	Associativity
1.	~	Bitwise NOT	Unary	Unary	Level-I	R ↱ L
2.	<< >>	Left Shift Right Shift	Shift Operators	Binary	Level-II	L ↱ R
3.	&	Bitwise AND	Bitwise Operator	Binary	Level-III	L ↱ R
4.	^	Bitwise X-OR	Bitwise Operator	Binary	Level-IV	L ↱ R
5.	|	Bitwise OR	Bitwise Operator	Binary	Level-V	L ↱ R

The bitwise-AND and the bitwise-OR operators operate on the individual bits of the operands according to the truth table specified above. X-OR operates according to the truth table given below:

X-OR Operation		
Operand1	Operand2	Result
False	False	False
False	True	True
True	False	True
True	True	False

The bitwise-NOT operator results in 1's complement of its operand. Left shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow. Right shift by n bits is equivalent to integer division by 2^n.

v) Assignment operators: A variable can be assigned a value by using an assignment operator. The assignment operators available in C language are given below:

S.No.	Operator	Name of Operator	Category	Type	Precedence Among Bitwise Class	Associativity
1.	=	Simple assignment	Assignment & Shorthand Assignment Operators	Binary	Level-I	R ⇗ L
	*=	Assign product				
	/=	Assign quotient				
	%=	Assign modulus				
	+=	Assign sum				
	-=	Assign difference				
	&=	Assign bitwise AND				
	!=					
	^=	Assign bitwise OR				
	<<=	Assign bitwise XOR				
	>>=	Assign left shift				
		Assign right shift				

The operand that appears towards the left side of an assignment operator should have a modifiable L-value. The shorthand assignment is of the form op1 op= op2, where op1 and op2 are operands and op= is a shorthand assignment operator. For example, a /= 2 is equivalent to a = a / 2. There should be no whitespace character between two symbols of shorthand assignment operators.

If two operands of an assignment operator are of different types, the type of operand on the right side of the assignment operator is automatically converted to the type of operand present on its left side. To carry out this conversion, either promotion or demotion is applied.Differences between initialization and assignment

Initialization	Assignment

First time assignment at the time of definition is called initialization. Example: `int a = 10;` is initialization of `a`.	Value of a data object after initialization can be changed by the means of assignment. Example: Consider the following statements
	`int a = 10;`
	`a = 20;`
	The value of `a` is changed to `20` by the assignment statement.

Initialization can be done only once.	Assignment can be done any number of times.
Qualified constant can be initialized with a value. Example: const `int a = 10;` is valid.	Qualified constant cannot be assigned a value. It is erroneous to write `a = 10;` if `a` is a qualified constant.

vi) Miscellaneous operators: Other operators available in C are:

a) Function Call Operator (i.e. ( ))

b) Array Subscript Operator (i.e. [ ])

c) Member select operator

d) Indirection Operator

(i) Direct Member Access Operator (i.e. . (dot operator or period))

(ii) Indirect Member Access Operator (i.e. -> (arrow operator))

e) Conditional Operator: Conditional operator is the only ternary operator available in C.

S.No.	Operator	Name of Operator	Category	Type	Precedence Among Bitwise Class	Associativity
1.	?:	Conditional operator	Conditional	Ternary	Level-I	R ⇗ L

The general form of condition operator is

E1 ? E2 : E3

where E1, E2 and E3 are sub-expressions. The sub-expression E1 is evaluated first. If it evaluates to a non-zero value (i.e. true), then E2 is evaluated and E3 is ignored. If E1 evaluates to zero (i.e. false), then E3 is evaluated and E2 is ignored.

f) Comma Operator: The comma operator is used to join multiple expressions together. The comma operator guarantees left-to-right evaluation. In expression E1, E2, E3, …, En are evaluated in left-to-right order. The result and type of evaluation of the overall expression is the value and type of the evaluation of the rightmost sub-expression (En). The comma operator has least precedence.

Example:	Reason:

#include<stdio.h>
main()
{
int a, b;
a = 1, 2, 3, 4, 5;
b = (1, 2, 3, 4, 5);
printf("The values of a and b are:\n");
printf("%d\t%d", a, b);
}
 	
Output:
 	
The values of a and b are:
1 5
	• The precedence of assignment operator is greater than comma operator.
	• Thus, in the expression a = 1, 2, 3, 4, 5, the sub-expression a = 1 gets evaluated first. Hence, the value assigned to a is 1.
	• In the expression b = (1, 2, 3, 4, 5), the sub-expression 1, 2, 3, 4, 5 is parenthesized and will be evaluated first. The result of evaluation of comma operator is the result of evaluation of the rightmost sub-expression, i.e. 5. Thus, (1, 2, 3, 4, 5) evaluates to 5 and is assigned to b.

g) sizeof Operator: The sizeof operator is used to determine the size in bytes, which a value or a data object will take in memory. The general form of a sizeof operator is:

(i) sizeof expression or sizeof (expression) Example: sizeof 2, sizeof(a), sizeof(2 + 3)

(ii) sizeof (type-name) Example: sizeof(int), sizeof(char)

The type of result of evaluation of the sizeof operator is int. The operand of the sizeof operator is not evaluated. This operator cannot be applied on operands of incomplete type or function type.

h) Address-of Operator: The address-of operator is used to find the address of a data object and must appear towards the left side of its operand. The operand of the address-of operator should be a variable or a function designator. This operator cannot be applied to constants, expressions, bit-fields and to the variables declared with register storage class.

S.No.	Operator	Name of Operator	Category	Type	Precedence	Associativity
1.	( ) [ ] -> .	Function call Array subscript Indirect member access Direct member access			Level-I (Highest)	
2.	! ~ + – ++ -- & * sizeof	Logical NOT Bitwise NOT Unary plus Unary minus Increment Decrement Address-of Dereference Sizeof	Unary operators	Unary	Level-II	R→L
3.	* / %	Multiplication Division Modulus	Multiplication operators		Level-III	L→R
4.	+ –	Addition Subtraction	Additive operators		Level-IV	L→R
5.	<< >>	Left shift Right shift	Shift operators		Level-V	L→R

6.	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-VI	L⇗R
7.	== !=	Equal to Not equal to	Equality operators		Level-VII	L⇗R
8.	&	Bitwise AND	Bitwise operator		Level-VIII	L⇗R
9.	^	Bitwise X-OR			Level-IX	L⇗R
10.	|	Bitwise OR			Level-X	L⇗R
11.	&&	Logical AND	Logical operator		Level-XI	L⇗R
12.	||	Logical OR			Level-XII	L⇗R
13.	?:	Conditional operator	Conditional	Ternary	Level-XIII	R⇗L
14.	= *= /= %= += -= &= |= ^= <<= >>=	Simple assignment Assign product Assign quotient Assign modulus Assign sum Assign difference Assign bitwise AND Assign bitwise OR Assign bitwise XOR Assign left shift Assign right shift	Assignment & Shorthand assignment operators	Binary	Level-XIV	R⇗L

15.	,	Comma operator	Comma		Level-XV	L⇗R

4.(i) Construct and explain the concept of operator precedence and associativity of operators with examples.(8)

– Precedence of Operators:

Each operator in C has a *precedence* associated with it. In a compound expression, if the operators used in the expression are of different precedence, the operator of higher precedence operates first.

For example,

in an expression *b* = 2 + 3 * 5, the sub-expression 3 * 5 is evaluated first as the multiplication operator has the highest precedence among =, + and *. In the resultant expression, the sub-expression 2 + 15 is evaluated next as the addition operator (i.e. +) has a higher precedence than the assignment operator (i.e. =). Finally, the assignment operator assigns the value 17 to *b*.

If two or more operators in a compound expression are of the same precedence, to determine which of these operators will operate first, the associativity of these operators is to be considered.

– Associativity of Operators:

The operators with the same precedence always have the same associativity. An operator can be either left-to-right associative or right-to-left associative. If operators are left-to-rightassociative, they are applied in a left-to-right order, and if they are right-to-left associative, they will be applied in the right-to-left order.

For example,

in the expression 2 * 3 / 5, the multiplication operator is evaluated prior to the division operator as it appears before the division operator in the left-to-right order.

4.(ii) Explain the bitwise operator with an example.(8)

Answered in IIIrd question 4th point of Bitwise Operator with example

5.Explain the following: (4+4+4+4=16)

i. Keywords ii. Identifiers iii. C character set iv. Expressions.

i. Keywords

Keyword is a reserved word that has a particular predefined meaning in the programming language. A keyword cannot be used as an identifier name in C language. There are 32 keywords available in C.

S.No.	Keyword	S.No.	Keyword	S.No.	Keyword	S.No.	Keyword
1.	auto	9.	double	17.	int	25.	struct
2.	break	10.	else	18.	long	26.	switch
3.	case	11.	enum	19.	register	27.	typedef
4.	char	12.	extern	20.	return	28.	union
5.	const	13.	float	21.	short	29.	unsigned
6.	continue	14.	for	22.	signed	30.	void
7.	default	15.	goto	23.	sizeof	31.	volatile
8.	do	16.	if	24.	static	32.	while

II. identifier

An identifier can be a variable name, a label name, a function name, a typedef name, a macro name or a macro parameter, a tag or a member of a structure, a union, or an enumeration. The rules to define an identifier in C are as follows:

1. Identifier name in C can have letters, digits or underscores.
2. The first character must be a letter (either uppercase or lowercase) or an underscore.
3. No special character (except underscore) can be used in an identifier name.
4. Keywords or reserved words cannot form a valid identifier name.
5. The maximum number of characters allowed in an identifier name is compiler dependent.

Student_Name, StudentName, student_name, student1, _student

type to the compiler before its use. The general form is:

[storage_class_specifier] [type_qualifier][type_modifier] type identifier [=value[…]];

Example: int variable, int variable=20; int a=20, b=10; unsigned int variable; const unsigned int variabl

III.C character set

A character set defines the valid characters that can be used in a source program or interpreted when a program is running. The set of characters that can be used to write a source program is called a source character set, and the set of characters available when the program is being executed in called an execution character set. The basic source character set of C language includes:

1. Letters: Uppercase letters: A, B, C, …, Z and lowercase letters: a, b, c, …, z
2. Digits: 0, 1, 2, …, 9
3. Special characters: , . : ; ! " @ # % ^ & * ( ) { } [ ] < > | / \ _ ~ etc.
4. White space characters: blank space character, horizontal tab space character, carriage return, new line character, form feed character

iv.Expression:

An expression in C is made up of one or more operands and operators that specify the computation of a value.

– **Operands**: An operand specifies an entity on which an operation is to be performed. An operand can be a variable name, a constant, a function call or a macro name.

– **Operators**: An operator specifies the operation to be applied to its operands.

Example: a = 2 + 3. Here, 2 and 3 are the operands, + and = are the operators. The operands 2 and 3 are added using the + operator and the value of the addition (i.e. 5) is assigned to the variable a using the assignment (i.e. =) operator.

Based on number of operators present in an expression, expressions are classified as **simple expressions** and **compound expressions**.

– **Simple Expressions**: An expression that has only one operator in it is known as a simple expression. **Example**: b = a + 2.

– **Compound Expressions**: An expression that involves more than one operator is called a compound expression. Example: b = 2 + 3 * 5.

6.(i) Describe the different type of format specifies.(8)

Formatted Functions

The formatted functions are scanf and printf functions.

1. scanf Function:

The scanf function is used to take the input from the user. The rules for using scanf function are as follows:

i) The name of scanf function should be in lowercase.

ii) The inputs (or arguments) to scanf function are given within parentheses.

iii) The first input to scanf function should always be a format string. Ideally, the format string of a scanf function should only consist of blank separated format specifiers as shown below.

S.No.	Data Type	Format Specifier	Remark
1.	char	%c	Single character
2.	int	%i	Signed integer
3.	int	%d	Signed integer in decimal number system
4.	unsigned int	%o	Unsigned integer in octal number system
5.	unsigned int	%u	Unsigned integer in decimal number system
6.	unsigned int	%x	Unsigned integer in hexadecimal number system
7.	unsigned int	%X	
8.	long int	%ld	Signed long
9.	short int	%hd	Signed short
10.	unsigned long	%lu	Unsigned long
11.	unsigned short	%hu	Unsigned short
12.	float	%f	Signed single precision float in fractional format
13.	float	%e	Signed single precision float in exponent format

14.	float	%E	Same as %e with E for exponent
15.	float	%g	Signed value in either %e or %f based on given value
16.	float	%G	Same as %g with E for exponent if e format is used
17.	double	%lf	Signed double-precision float
18.	String type	%s	String
19.	Pointer type	%p	Pointer

iv) The inputs are separated by commas.

v) The inputs following the first input should denote variables. The symbol & is address-of operator and is used to find the L-value of its operand.

Example: **scanf("%d",&number);** Here, &number refers to the L-value and the variable number should be declared of type int.

Reading Strings from the Keyboard: The user can enter strings and store them in character arrays at the runtime by using the following methods:

i) Using scanf function:

The scanf function with %s format specifier can be used to read a string from the user. This function can be used to read only single word strings. The scanf function automatically terminates the input string with a null character (i.e. \0), and therefore the character array should be large enough to hold the input string plus the null character.

It is not mandatory to use ampersand (&) , i.e. address-of operator, with string variable names while reading strings using the scanf function, since the string variable is a character array and the name of an array refers to the address of the first element of the array.

Example: **scanf("%s", name);**

where name is a character array (i.e. char name[30]). Here, if the usergives "Sam Mine" as input, the string "Sam" is assigned to name, because the input contains multiple words (i.e. Sam and Mine).

The scanf function can be used to read a specific number of characters by specifying the field width.

Example: **scanf("%3s", name);** Here, if the user gives "Samuel" as input, the string "Sam" is assigned to name, because if the length of the entered string is more than the specified field width (i.e. 3), the number of characters read will be at most equal to the field width (i.e. 3).

The scanf function can also be used to read selected characters by using search sets. A search set defines a set of possible characters that can make up the string and it is case sensitive. For example, a) scanf("%[abcd]",name); Here, if the user gives "daman" as input, the string "da" is assigned to

name, because the scanf reads the input characters and stops when a character except a, b, c or d is encountered.

b) **scanf("%[^abcd], name);** Here, if the user gives "mechanical" as input, the string "me" is assigned to name, because the search set is inverted to include all the characters (even white-space characters) except those between the brackets (i.e. a, b, c, d).

2. **printf Function**: The printf function is used to print the output in the display. The rules for using printf function are as follows:

   i) The name of printf function should be in lowercase.

   ii) The inputs (or arguments) to printf function are given within parentheses.

   iii) At least one input is required, and the first input to printf function should always be a string literal.

   iv) The inputs are separated by commas.

   v) If values of identifiers are to be printed with the help of printf function, the first input to printf function should be a format string. A format string consists of format specifiers and they specify the format according to which the printing will be done.

   Example: printf("The value is %d", num); It prints "The value is 20". Here the variable num should be declared of type int and it is assumed that num = 20.

Unformatted Functions

C has three types of unformatted I/O functions.

1. **Character I/O**
2. **String I/O**
3. **File I/O**