# MODULE 1: INTRODUCTION TO C LANGUAGE

## Questions and Answers on Pseudocode solution to a problem

## Q.1 What is algorithm? Write the characteristics of an algorithm. Give an example for algorithm.

## Ans:

**Definition of Algorithm**

Algorithm is a step by step solution to a given problem. Sequence of steps written in order to carry out some particular task. Each step of an algorithm is written in english.

**Various charaterstics of an algorithm are**

1. The algorithm must have definite start and end.
2. An algorithm may accept zero or more inputs
3. An algorithm must produce atleast one output
4. The steps of an algorihtm must be simple, easy to understand and unambigous.
5. Each step must be precise and accurate.
6. The algorihtm must contain finite number of steps.

**Example:**

**Algorithm 1: Algorihtm for finding of area of traingle**

Step 1: start
Step 2: Read base and height of traingle
Step 3: Calulate area of traingle
Step 4: print area of traingle
Step 5: stop

**Algorithm 2: Algorihtm for finding of sum and average of given three numbers**

Step 1: start
Step 2: Read three numbers i.e. A, B and C
Step 3: find sum of three numbers i.e. sum=A+B+C
Step 4: find average of three numbers i.e. average=sum/3
Step 4: print sum and average
Step 5: stop

## Q. 2 What is pseudocode? Give an example for pseudocode.

## Ans:

**Definition of Pseudocode**

Pseudocode is high level description of an algorithm that contains a sequence of steps written in combination of english and mathemetical notations to solve a given problem.

Pseudocode is part english and part program logic.

Pseudocodes are better than algorithm since it contains ordered steps and mathemetical notations they are more closer to the statements of programming language.

This is essentially an intermediate-step towards the development of the actual code(program). Although pseudo code is frequently used, there are no set of rules for its exact writing.

**Example:**

**Pseudocode 1: Pseudocode for finding area of traingle**

```
Pseudocode Area_of_Traingle
BEGIN
        READ base and Height
        CALCULATE Area_of_Triangle=(base*height)/2
        PRINT Area_of_Triangle
END
```

**Pseudocode 2:Pseudocode for finding sum and average of three numbers**

```
Pseudocode SUM_AVG
BEGIN
        READ A, B, and C
        CALCULATE sum=A+B+C
        CALCULATE average=SUM/3
        PRINT sum and average
END
```

## Q.3 What is flowchart? List and define the purpose of symbols used to represent flowchart. Give an example for flowchart.

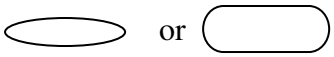
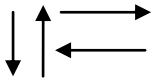
## Ans:

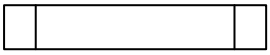
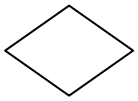
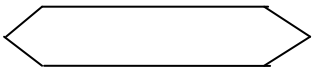
**Defintion of flowchart**

Flowchart is a graphical or pictorial representation of an algorithm. Its a program design tool in which standard graphical symbols are used to represent the logical flow of data through a function.

Flowchart is a combination of symbols. They show start and end points, the order and sequence of actions, and how one part of a flowchart is connected to another.

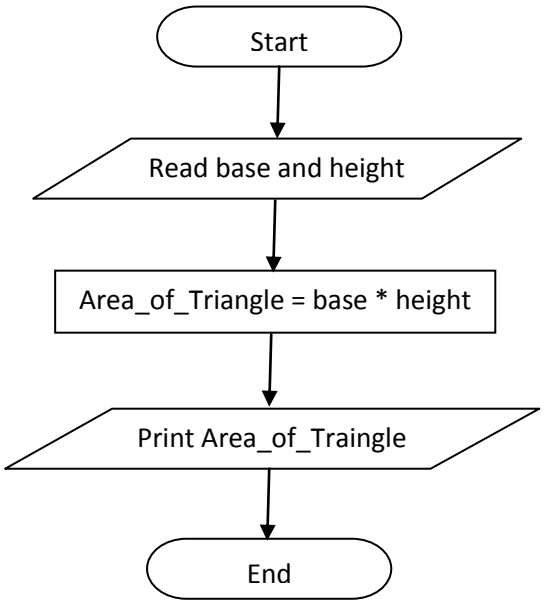
**The symbols and thier purpose in flowchart is given below.**

| Symbol | Name | Purpose |
|---|---|---|
|  | Oval or Rounded Rectangle | Shows the Start or End of an algorithm |
|  | Arrows | Shows the Flow of data |
|  | Connector | Shows connecting points in an algorithm |
|  | Paralellogram | Used to indicate Input or Ouput statement |

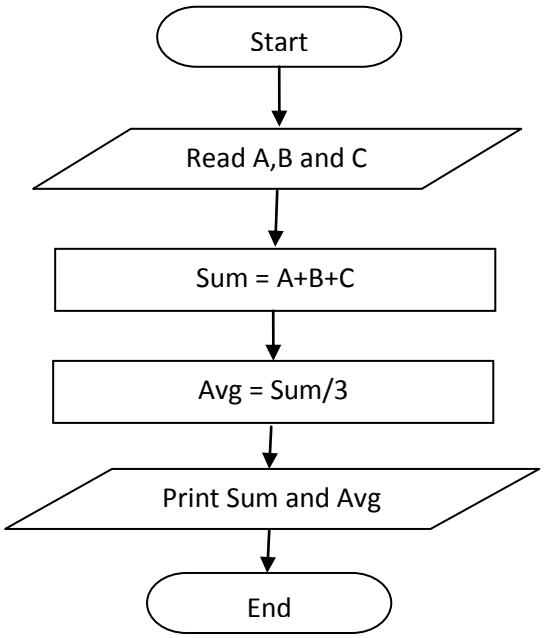
| | | |
|---|---|---|
|  | Rectangle | Used to indicate assignment statement and executable statement |
|  | Module Call | Used to represent function call |
|  | Rhombus | Used to represent decision making statement |
|  | Repitation | Used to represent looping statement |

Example:

**Flowchart 1: Finding the area of traingle**




**Flowchart 2: Finding sum and average of given three numbers**




## Questions and Answers on Basic Concepts in a C Program

### Q.4 What are the *alphabets or character set* of C programming language?

**Ans:**

The characters that can be used to form words, numbers and expressions.
The characters in C are grouped into the following categories.

- **Letters:** Lowercase letters a,b,...,z and uppercase letters A,B,...,Z
- **Digits:** from 0 to 9
- **White spaces:** Space, tab(\t), newline(\n), carriage return(\r), etc
- **Symbols:**

| | | |
|---|---|---|
| ~ tilde | ! bang | % modulus |
| & ampersand | * star or asterisk | / division |
| ( left parentheses | { left brace | [ left bracket |
| ) right parentheses | } right brace | ] right bracket |
| \| vertical bar | ^ caret | ‘ single quote |
| “ double quotation | < less than | > greater than |
| : colon | ; semicolon | \ back slash |
| . dot | , comma | ? question mark |
| # hash or pound sign | _ underscore | = assignment |

### Q.5 What are *C-Tokens*? List and define different *C-Tokens* with examples.

**Ans:**

A C-Token is a smallest element of a C program. One or more characters are grouped in sequence to form meaningful words. These meaningful words are called C-Tokens.
The tokens are broadly classified as follows

• **Keywords**: Keywords are tokens which are used for their intended purpose only. Each keyword has fixed meaning or predefined meaning and that cannot be changed by user. Hence, they are also called as **reserved-words**.

ex: if, for, while, int, float, break, char ..etc.

• **Identifiers:** As the name indicates, identifier is used to identify various elements of program such as variables, constants, functions, arrays etc.

ex: sum, length, etc.

• **Constants**: A constant is an identifier whose value remains fixed throughout the execution of the program. The constants cannot be modified in the program.

ex: 10, 10.5, 'a', "sri", etc.

• **Operators**: An operator can be any symbol like + - * / that specifies what operation need to be performed on the data.

For ex: + indicates addition operation

* indicates multiplication operation

/ indicates division operation, etc.

• **Special symbols**: The symbols that are used to indicate array subscript, function call, block, etc.

ex: [], (), { }, etc.

## Q. 6 What are *Keywords*? List all keywords and thier purpose/importance.

## Ans:

Keywords are tokens which are used for their intended purpose only. Each keyword has fixed meaning or predefined meaning and that cannot be changed by user. Hence, they are also called **reserved-words**.

There are totally 32 keywords. All keywords are written in lowercase letters.

**auto** : The auto keyword declares automatic variables.Variables declared within function bodies are automatic by default.

**break and continue:** The *break* statement makes program jump out of the innermost enclosing loop (while, do, for or switch statements) explicitly. The *continue* statement skips the certain statements inside the loop.

**switch, case and default :** The switch and case statement is used when a block of statement has to be executed among many blocks/alternatives.

**char :** The char keyword declares a character variable.

**const:** An identifier can be declared constant by using const keyword.

**do...while**: do...while are used to repeat set of statements

**double and float:** Keywords double and float are used for declaring floating type variables.

**if and else:** if and else are used to make decisions.

**enum:** Enumeration types are declared in C programming using keyword enum.

**extern:** The extern keyword declares that a variable or a function has external linkage outside of the file it is declared.

**for:** for is used to repeat set of statements

**goto:** The goto keyword is used for unconditional jump to a labeled statement inside a function.

**int**: The int keyword declares integer type variable.

**short, long, signed and unsigned:** The short, long, signed and unsigned keywodrs are type modifiers that alters the meaning of a base data type to yield new type.

**return:** The return keyword terminates the function and returns the value.

**sizeof:** The sizeof keyword evaluates the size of a data (a variable or a constant).

**register:** The register keyword creates register variables which are much faster than normal variables.

**static:** The static keyword creates static variable. The value of the static variables persists until the end of the program.

**struct:** The struct keyword is used for declaring a structure. A structure can hold variables of different types under a single name.

**typedef:** The typedef keyword is used to define user defined name for a datatype.

**union:** A Union is used for grouping different types of variable under a single name for easier handling.

**void:** The void keyword indicates that a function doesn't return value.

**volatile:** The volatile keyword is used for creating volatile objects. A volatile object can be modified in unspecified way by the hardware.

## Q. 7 What are identifiers? What are the rules for naming a identifier? Give examples.

## Ans:

Identifier is used to name various elements of program such as variables, constants, functions, arrays, functions etc. An identifier is a word consisting of sequence of Letters, Digits or _(underscore).

**Rules to Name identifiers** are

1. The variables must always begin with a letter or underscore as the first character.
2. The following letters can be any number of letters, digits or underscore.
3. Maximum length of any identifier is 31 characters for external names or 63 characters for local names.
4. Identifiers are case sensitive. Ex. Rate and RaTe are two different identifiers.
5. The variable name should not be a keyword.

6. No special symbols are allowed except underscore.

Examples:

**Valid identifiers:**

Sum roll_no _name sum_of_digits avg_of_3_nums

**Invalid Identifiers and reason for invalid:**

| | | |
|---|---|---|
| $roll_no | - | Name doesn't begin with either letter or underscore |
| for | - | keyword is used as name |
| sum,1 | - | special symbol comma is not allowed |
| 3_factorial | - | name begins with digit as first letter |

## Q. 8 What are constants? List and explain different types of constants with examples.

## Ans:

A constant is an identifier whose value remains fixed throughout the execution of the program. The constants cannot be modified in the program.

For example: 1, 3.14512, 'z', "hello"

**Different types of constants** are:

1) **Integer Constant:** An integer is a whole number without any fraction part.
   There are 3 types of integer constants:
   i) Decimal constants (0 1 2 3 4 5 6 7 8 9) For ex: 0, -9, 22
   ii) Octal constants (0 1 2 3 4 5 6 7) For ex: 021, 077, 033
   iii) Hexadecimal constants (0 1 2 3 4 5 6 7 8 9 A B C D E F)
   For ex: 0x7f, 0x2a, 0x521
2) **Floating Point Constant**: The floating point constant is a real number.
   The floating point constants can be represented using 2 forms:
   i) **Fractional Form**: A floating point number represented using fractional form has an integer part followed by a dot and a fractional part.
   For ex: 0.5, -0.99
   ii) **Scientific Notation (Exponent Form)**: The floating point number represented using scientific notation has three parts namely: mantissa, E and exponent.
   For ex: 9.86E3 imply $9.86 \times 10^3$
3) **Character Constant**: A symbol enclosed within a pair of single quotes(') is called a character constant.
   Each character is associated with a unique value called an ASCII (American Standard Code for Information Interchange) code.
   For ex: '9', 'a', '\n'
   ASCII code of A = 65, a= 97, 0=48, etc
4) **String Constant**: A sequence of characters enclosed within a pair of double quotes(") is called a string constant.
   The string always ends with NULL (denoted by \0) character.
   For ex: "9", "a", "sri", "\n", ""

5) **Escape Sequence Characters**: An escape sequence character begins with a backslash and is followed by one character.

• A backslash (\) along with some character give special meaning and purpose for that character.

• The complete set of escape sequences are:

- \b Backspace
- \f Form feed
- \n Newline
- \r Return
- \t Horizontal tab
- \v Vertical tab
- \\ Backslash
- \' Single quotation mark
- \" Double quotation mark
- \? Question mark
- \0 Null character

6) **Enumeration constants:** It is used to define named integer constants. It is set of named integer constants.

**Syntax for defining enumeration constants**

enum identifier { enumeration-list}; // enum is a keyword

Example:

1) enum boolean {NO, YES};

/* This example defines enumeration boolean with two constants, NO with value 0 and YES with value 1 */

In enumeration-list, the first constant value i.e. NO is ZERO since it is not defined explicitly, and then value of next constant i.e. YES is one more than the previous constant value(YES=1).

2) enum months {Jan=1, Feb, Mar, Apr, May, June,Jul,Aug,Sep,Oct,Nov,Dec};

/* This example define enumeration months with 12 constants, Jan=1,feb=2,mar=3,apr=4, ..., Dec=12 */

In enumeration-list, the first constant value i.e. Jan=1 since it is defined explicitly, and then value of next constants i.e. Feb, Mar, Apr,...,Dec are one more than the previous constant value.

## Q.9 What is datatype? List and explain different datatypes with examples.

## Ans:

Datatype is an entity that defines set of values and set of operations that can be applied on those values.

Datatypes are broadly categorized as TWO types:

1) **Primitive Data Types**: Fundamental data types or Basic data types are primitive data types.

**Examples**: int, char, float, double, void

2) **Non-Primitive Data Types**: The derived or user-defined data types are called as non-primitive data types.

**Examples**: struct, union, pointers, etc.

- **int** type:

The int type stores integers in the form of "whole numbers". An integer is typically the size of one machine word, which on most modern home PCs is 32 bits. Examples of whole numbers (integers) such as 1,2,3, 10, 100... When int is 32 bits, it can store any whole number (integer) between -2147483648 and 2147483647. A 32 bit word (number) has the possibility of representing any one number out of 4294967296 possibilities (2 to the power of 32).

```
int numberOfStudents, i, j=5;
```

In this declaration it declares 3 variables, numberOfStudents, i and j, j here is assigned the literal 5.

- **char** type

The char type is capable of holding any member of the character set. It stores the same kind of data as an int (i.e. integers), but typically has a size of **one byte**. The size of a byte is specified by the macro CHAR_BIT which specifies the number of bits in a char (byte). In standard C it never can be less than 8 bits. A variable of type char is most often used to store character data, hence its name.

Examples of character literals are 'a', 'b', '1', etc., as well as some special characters such as '\0' (the null character) and '\n' (newline, recall "Hello, World"). Note that the char value must be enclosed within single quotations.

```
char letter1 = 'a';      /* letter1 is being initialized with the letter 'a' */
char letter2 = 97;       /* in ASCII, 97 = 'a' */
```

In the end, letter1 and letter2 both stores the same thing the letter 'a', but the first method is clearer, easier to debug, and much more straightforward.

- **float** type

float is short for floating point. It stores real numbers also, but is only one machine word in size. Therefore, it is used when less precision than a double provides is required. float literals must be suffixed with F or f, otherwise they will be interpreted as doubles. Examples are: 3.1415926f, 4.0f, 6.022e+23f. float variables can be declared using the float keyword.

- **double** type

The double and float types are very similar. The float type allows you to store single-precision floating point numbers, while the double keyword allows you to store double-precision floating point numbers – real numbers, in other words, both integer and non-integer values. Its size is typically two machine words, or 8 bytes on most machines. Examples of double literals are 3.1415926535897932, 4.0, 6.022e+23 (scientific notation).

- **void** type

It is an empty data type. It has no size and no value. It is used with functions which doesnot return any value, pointers,etc.

**Range of values for char, int data type can be calculated with following formula**

1) **Range of Signed numbers (Both Positive and Negative numbers)**

$-2^{n-1}$ to $+2^{n-1}-1$ where n is number of bits

Therefore,

- char - 1 byte (8 bits)
  range of values = $-2^{8-1}$ to $+2^{8-1}-1$ => -128 to +127
- int - 2 bytes (16 bits in 16-bit OS)
  range of values= $-2^{16-1}$ to $+2^{16-1}-1$=>-32768 to +32767

2) **Range of UnSigned numbers ( Only Positive Numbers)**

0 to $+2^{n-1}-1$ where n is number of bits

Therefore,

- char - 1 byte (8 bits)

  range of values = 0 to $+2^{8}-1$ => 0 to 255

- int - 2 bytes (16 bits in 16-bit OS)

  range of values= 0 to $+2^{16}-1$=>0 to +65535

**Size of each data type and ranges of values is given below.**

| | | | |
|---|---|---|---|
| char | | 1 byte ( 8 bits ) | range -128 to 127 |
| int | 16-bit OS : | 2 bytes | range -32768 to 32767 |
| | 32-bit OS : | 4 bytes | range -2,147,483,648 to 2,147,483,647 |
| float | | 4 bytes | range 3.4E-38 to 3.4E+38 with 6 digits of precision |
| double | | 8 bytes | range 1.7E-308 to 1.7E+308 with 15 digits of precision |

void generic pointer, used to indicate no function parameters, no retrun value etc.

**Type Modifiers:** Except for type void the meaning of the above basic types may be altered when combined with the following keywords.

- signed
- unsigned
- long
- short

**Table: Use of modifiers to create modified data type**

| Primitive Data Type | Type Modifier | Modified Data Type |
|---|---|---|
| Char | Unsigned<br>signed | unsigned char<br>signed char |
| Int | Unsigned<br>signed<br>short<br>long | unsigned int<br>signed int<br>short int<br>short<br>unsigned short int<br>unsigned short |

| | | |
|---|---|---|
| | | signed short int<br>signed short<br>long int<br>long<br>unsigned long int<br>unsigned long<br>signed long int<br>signed long |
| Float | N/A | N/A |
| Double | Long | long double |

The **signed and unsigned** modifiers may be applied to types **char and int** and will simply change the range of possible values.

For example an unsigned char has a range of 0 to 255, all positive, as opposed to a signed char which has a range of -128 to 127.

An unsigned integer on a 16-bit system has a range of 0 to 65535 as opposed to a signed int which has a range of -32768 to 32767.

Note however that the default for type int or char is signed so that the type signed char is always equivalent to type char and the type signed int is always equivalent to int.

The **long modifier** may be applied to type int and double only. A long int will require 4 bytes of storage no matter what operating system is in use and has a range of -2,147,483,648 to 2,147,483,647.

A long double will require 10 bytes of storage and will be able to maintain up to 19 digits of precision.

The **short modifier** may be applied only to type int and will give a 2 byte integer independent of the operating system in use.

**Examples:**

short int, long int, long double, unsigned char, signed char, unsigned int, etc

## Q.10 Give general structure of C program. Explain with an example program.

## Ans:

A C program is essentially a group of instructions that are to be executed as a unit in a given order to perform a particular task.

Each C program must contain a main() function. This is the first function called when the program starts to run or execute.

A C program is traditionally arranged in the following order but not strictly as a rule.

```
/* Comment lines */
Preprocessor Directives
[Global Declaration]
int main()
{
        [Local Declarations]
        [Executable statements]
```

**}**

**[User-defined Functions]**

**Example Program:**

Consider first a simple C program which simply prints a line of text to the computer screen. This is traditionally the first C program you will see and is commonly called the "Hello World" program for obvious reasons.

```
#include <stdio.h>
 int main()
{
        /* This is how comments are implemented in C      to comment out a block of text */
        // or like this for a single line comment
        printf( "Hello World\n" ) ;
        return 0;
 }
```

All C compiler include a library of standard C functions such as printf which allow the programmer to carry out routine tasks such as I/O operations, mathemetical operations, string operations etc. but which are not part of the C language, the compiled C code merely being provided with the compiler in a standard form.

Header files must be included which contain prototypes for the standard library functions and declarations for the various variables or constants needed. These are normally denoted by a .h extension and are processed automatically by a program called the **Preprocessor** prior to the actual compilation of the C program.

Therefore, The line **#include <stdio.h>**

Instructs the pre-processor to include the file stdio.h into the program before compilation so that the definitions for the standard input/output functions including printf will be present for the compiler.

As you can see this program consists of just one function the mandatory **main** function. The parentheses, ( ), after the word main indicate a function while the curly braces, { }, are used to denote a block of code -- in this case the sequence of instructions that make up the function.

**Comments** are contained within a /* ... */ pair in the case of a block(mutli-line) comment or a double forward slash, //, may be used to comment out single line.

The line **printf("Hello World\n " ) ;**

is the only C statement in the program and must be terminated by a semi-colon. The statement calls a function called printf which causes its argument, the string of text within the quotation marks, to be printed to the screen. The characters \n are not printed as these characters are interpreted as special characters by the printf function in this case printing out a newline on the screen. These characters are called escape sequences in C and cause special actions to occur and are preceded always by the backslash character, \ .

**Global and Local Declaration** statements are used to declare global and local variables, arrays, functions, pointers, etc.

## Questions and Answers on Declaration, Initialization and Print statements

## Q. 11 What is Variable? What is the need for variables? What are the rules for naming variables. Give examples.

## Ans:

**Definition of Variable**

Variable is an identifier used to name the memory location which holds the value. Variable is an entity whose value can be changed (Not Fixed) during the program execution.

**Need for variables**

It may help to think of variables as a placeholder for a value or data. You can think of a variable as being equivalent to its assigned value or data. In a C program, if a user wants to store and use any data then the user must create variables in a program to store the required data.

**Rules for naming a Variable**

1. The variables must always begin with a letter or underscore as the first character.
2. The following letters can be any number of letters, digits or underscore.
3. Maximum length of any identifier is 31 characters for external names or 63 characters for local names.
4. Identifiers are case sensitive. Ex. Rate and RaTe are two different identifiers.
5. The variable name should not be a keyword.
6. No special symbols are allowed except underscore.

Examples:

**Valid identifiers:**

Sum roll_no _name sum_of_digits avg_of_3_nums

**Invalid Identifiers and reason for invalid:**

| | | |
|---|---|---|
| $roll_no | - | Name doesn't begin with either letter or underscore |
| for | - | keyword is used as name |
| sum,1 | - | special symbol comma is not allowed |
| 3_factorial | - | name begins with digit as first letter |

## Q.12 What is Variable? How to Declare and initialize variables? Give examples.

## Ans:

**Definition of Variable**

Variable is an identifier used to name the memory location which holds the value. Variable is an entity whose value can be changed (Not Fixed) or modified during the program execution.

**Declaring a Variable**

A variable thus has three attributes that are of interest to us: its **type**, its **value** and its **address** and before a C program can utilize memory to store a value or data it must claim the memory needed to store the values for a variable. This is done by **declaring variables.**

Declaring variables is the way in which a C program shows the number of variables it needs, name of the variables, range of values it can represent and how much memory they need.
Within the C programming language, when managing and working with variables, it is important to know the **type of variables** and **the size of these types**. Size of the datatypes can be hardware specific – that is, how the language is made to work on one type of machine can be different from how it is made to work on another.
All variables in C are typed. That is, every variable declared must be assigned with certain datatype.

**General Syntax for declaring variables:**

DataType Variable-list;

Where,

DataType can be => int, float, char, double, short, unsigned int, long double, etc

Variable-list can be => list of variables seperated by comma.

DataType var1,var2,var3,...,varN;

**Example:**

```
int count;      /* It declares a variable count as integer type        */
float average;  /* It declares a variable average as floating point type */
int number_students, i, j; /* In this declaration it declares 3 variables, number_students, i and j. */
char code;      /* It declares a variable code as character type */
```

**Variables can be declared as**

1) **Local variables:**

When variables are declared inside functions as follows they are termed local variables and are visible (or accessible) within the function ( or code block ) only.

For Example,

```
int main()
{
    int i, j;
    ...
}
```

In the above example, the variables i and j are local variables, they are created i.e. allocated memory storage upon entry into the code block main() and are destroyed i.e. its memory is released on exit from the block. Therefore i and j are local to main().

2) **Global Variables:**

When variables declared outside functions they are termed as global variables and are visible throughout the program. These variables are created at program start-up and can be used for the entire lifetime of the program.

For Example,

```
int i;
int main()
{  ...  }
```

In this example, i is declared as global variable and it is visible throghout the program.

**Initialization or Assignment of a Variable**

When variables are declared in a program it just means that an appropriate amount of memory is allocated to them for their exclusive use. This memory however is not initialised to zero or to any other value automatically and so will contain random values unless specifically initialised before use. Hence, variables may need to be assigned and store value in it. It is done using initialzation.

**General Syntax for initialization :-**

Initialization of variable is done using assignement operator(=).

**variable-name = expression ;**

**Where,**

Expression can be - arithemetic expression, relational expression, logical expression, conditional expression, constant or an identifier that must reduce to a single value.

**For Example :-**

1) char ch;
   ch = 'a' ; /* in this example, a character constant ‘a’ is assigned to variabla ch */
2) double d;
   d = 12.2323 ; /* in this example, a floating point constant is initialized to d */
3) int i, j,k;
   i = 20 ;
   j=100;
   k=i+j; /* in this example, i is initialized to 20, j is initialized to 100 and k is initialized to addition of i and j i.e. 120 */

**Combining declaration and initialization of variables**

The variables can be declared and initialized at the time of creating i.e allocating memory storage for variables.

**Syntax** :- **DataType var-name = constant ;**

For Example :-

```
char ch = 'a' ;
double d = 12.2323 ;
int i, j = 20 ; /* note in this case  i is not initialised */
```

## Q.13 What are input and output functions? Explain printf() and scanf() functions with examples.

## Ans:

**Input Functions**: The functions which helps user to feed some data into program are called as input functions. When we are saying Input that means to feed some data into program. This can be given in the form of file or from command line or from keyboard. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Examples: scanf(), gets(), getchar(), fscanf(), etc

**Output Functions**: The functions which helps user to display or print output on screen or on paper using printer or in file are called as output functions. When we are saying Output that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen as well as you can save that data in text or binary files.

Examples: printf(), puts(), putchar(), fprintf(), etc.

## printf(): print formatted

The printf() function is used for formatted output and uses a control string or format string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required.

**General Syntax:**

**int printf(const char *format, ...);**

Printf function writes output to the standard output stream stdout(computer screen) and produces output according to a format provided.

Printf function returns an integer constant which is the number of character displayed on screen successfully. **The return value can be ignored.**

Two ways to use printf()

1) N=printf("Text string");
2) N=printf("format string",variables-list);

Where,

Text string - text message to be displayed on screen. It is displayed as it is on screen.

Format string - it is a sequence of one or more format specifiers depending on type of data to be displayed.

Variable-list - list of one or more variables, expressions, or constants whose data will be displayed on screen

N - contains the return value of printf function. It is optional means that it can be omitted.

The more **common format specifiers** for displaying various types of data are given below

| | |
|---|---|
| %c character | %f floating point |
| %d signed decimal integer | %lf double floating point |
| %i signed integer | %e exponential notation |
| %u unsigned integer | %s string |
| %ld signed long | %x unsigned hexadecimal |
| %lu unsigned long | %o unsigned octal |
| %% prints a % sign | |

For Example :-

```
int i ;
printf( "%d", i ) ;
```

In the above example, the format string is %d and the variable is i. The value of i is substituted for the format specifier %d which simply specifies how the value is to be displayed, in this case as a signed integer.

```
int N;
N = printf("Hello\n");
```

In this example, the printf() prints the text string "Hello" on screen and returns the integer value 5 which will be stored in N.

Some more examples :-

```
int i = 10, j = 20 ;
char ch = 'a' ;
double f = 23421.2345 ;
printf( "%d + %d", i, j ) ;  /* values of i and j are substituted from the variable list in order as required  */
printf( "%c", ch ) ;
printf( "%s", "Hello World\n" ) ;
printf( "The value of f is : %lf", f ) ;  /*Output as : The value of f is : 23421.2345 */
printf( "f in exponential form : %e", f ) ; /* Output as : f in exponential form : 2.34212345e+4
```

## Scanf(): scan formatted

The scanf() function is used for formatted input and uses a control string or format string which is made up of a series of format specifiers to govern how to read out the values for the variables as required.

This function is similar to the printf function except that it is used for formatted input.

The space character or the newline character are normally used as delimiters between different inputs.

**General Syntax of scanf():**

int scanf("format string",Address-list);

where,

format string - it is a sequence of one or more format specifiers to read data as required.

Address-list - it is a list of address of one or more variables depending on number of format specifiers.

**The format specifiers have the same meaning as for printf().**

The more **common format specifiers** for reading various types of data are given below

| | |
|---|---|
| %c character | %f floating point |
| %d signed decimal integer | %lf double floating point |
| %i signed integer | %e exponential notation |
| %u unsigned integer | %s string |
| %ld signed long | %x unsigned hexadecimal |
| %lu unsigned long | %o unsigned octal |

**For Example :-**

```
int i, d ; char c ; float f ;
scanf( "%d", &i ) ;
scanf( "%d%c%f", &d, &c, &f ) ; /* e.g. type "10 x 1.234" */
scanf( "%d:%c", &i, &c ) ;              /* e.g. type "10:x"  */
```

The & character is the address of operator in C, it returns the address of the memory location of the variable.

Note that while the space and newline characters are normally used as delimiters between input fields the actual delimiters specified in the format string of the scanf statement must be reproduced at the keyboard faithfully as in the case of the last example. If this is not done the program can produce somewhat erratic results!

**The scanf function has a return value which represents the number of fields it was able to convert or read successfully. It can be ignored.**

For Example :- num = scanf( "%c %d", &ch, &i );

This scanf function requires two fields, a character and an integer, so the **value placed in num** after the scanf() call will be 2 if successful.

## Questions and Answers on Operators and Expressions

## Q.14 What are operators and expressions? Explain different types of operators and expressions with examples.

**Ans:**

One of the most important features of C is that it has a very rich set of built in operators including arithmetic, relational, logical, and bitwise operators.

• An **operator** can be any symbol like + - * / that specifies what operation need to be performed on the data.

For ex: + indicates addition operation

* indicates multiplication operation

• An **operand** can be a constant or a variable.

• An **expression** is combination of operands and operators that reduces to a single value.

For ex: Consider the following expression a + b here a and b are operands, while + is an operator.

**Types of Operators and Expressions are:**

1) Arithmetic Operators and Expressions
2) Assignment and Compound Assignment Operators
3) Increment and Decrement Operators
4) Relational Operators and Expressions
5) Logical Operators and Expressions
6) Conditional Operators and Expressions
7) Bitwise Operators and Expressions
8) Special Operators

**1) Arithmetic Operators and Expressions:**

Arithmetic operators are + - * / and %

Arithmetic expressions are expressions which contains only arithmetic operators in it.

+ performs addition - performs subtraction * performs multiplication

/ performs division operation

% modulus or reminder operation

For Example:-

```
int a = 5, b = 2, x ;
float c = 5.0, d = 2.0, f ;
x = a / b ; //  integer division(5/2), therefore, x = 2.
f = c / d  ; //  floating point division(5.0/2.0), f = 2.5.
x = 5 % 2 ; //  remainder operator, x = 1.
x = 7 + 3 * 6 / 2 - 1 ;    // x=15,* and / evaluated ahead of + and -.
x = 7 + ( 3 * 6 / 2 ) - 1 ; // x = 15
x = ( 7 + 3 ) * 6 / ( 2 - 1 ) ; // changes order of evaluation, x = 60 now.
```

2) **Assignment Operators:** Used to initialize the variables with value.

Assignment operators is =

Compound assignment operators are +=,-=,/=,*=,%=. These are also called as shorthand operators.

Many C operators can be combined with the assignment operator as shorthand notation

For Example :- x = x + 10 ; can be replaced by x += 10 ;

Similarly for -=, *=, /=, %=, etc.

These shorthand operators improve the speed of execution as they require the expression, the variable x in the above example, to be evaluated once rather than twice.

3) **Increment and Decrement Operators**

There are two special unary operators in C, Increment ++, and Decrement -- , which cause the variable they act on to be incremented or decremented by 1 respectively.

For Example :-
```
x++ ;     /* equivalent to  x = x + 1 ;   */
x-- ;     /* equivalent to x = x-1; */
```

++ and -- can be used in Preincrement/Predecrement Notation( Prefix Notation) or Postincrement/Postdecrement Notation(Postfix notation).

In **Preincrement/Predecrement Notation**

a) The value of the variable is either incremented or decremented first
b) Then, will use updated value of the variable

But in **Postincrement/Postdecrement Notation**

a) The value of the variable is used first
b) Then the value of the variable is incremented or decremented

For Example :-
```
int i,  j = 2 ;
i = ++ j  ; /* preincrement: therefore i has value 3, j has value 3  */
int i,  j = 2 ;
i = j++ ; /* postincrement: therefore i  has value 2, j has value 3   */
```

4) **Relational Operators:** The operators which are to compare or to check the relation between two or more quantities.

Relational Expressions are expressions that contains only relational operators.

The full set of relational operators are provided in shorthand notation

| | | | |
|---|---|---|---|
| > | is greater than | >= | is greater than or equal to |
| < | is less than | <= | is less than or equal to |

== is equal to != is not equal to

The result of any relational expression is always either TRUE or FALSE.

Example 1:-

```
int x=2;
if ( x == 2 )     /* x ==2 is TRUE since x value is 2 */
        printf( "x is equal to 2\n" ) ;   // prints x is equal to 2
else
         printf( "x is not equal to 2\n" ) ;
```

Example 2:-

```
int x=5;
if ( x == 2 )     /* x ==2 is False since x value is 5 */
        printf( "x is equal to 2\n" ) ;
else
         printf( "x is not equal to 2\n" ) ;       // prints x is not equal to 2
```

5) **Logical Operators:** The operators which acts on only two logical values i.e. true and false.

Logical expressions are expressions that contains only logical operators. Usually to combine one or more relations these logical operators are used. Hence such expressions are called as relational logical expressions.

&& Logical AND || Logical OR

! Logical NOT

NOTE: Since C has no boolean datatype so to use these operators one must need to remember that

- ZERO(0) is always treated as FALSE and vice versa
- NON-ZERO (>0 or <0) value is always treated as TRUE. TRUE is always represented as 1.

For Example :-

```
if (  x >= 0 && x < 10  )
        printf( " x is greater than or equal to zero and less than ten.\n" ) ;
```

For Example :- 2 > 1 -- TRUE so expression has value 1

2 > 3 -- FALSE so expression has value 0

i = 2 > 1 ; -- relation is TRUE -- has value 1, i is assigned value 1

NOTE: Every C expression has a value. Typically we regard expressions like 2 + 3 as the only expressions with actual numeric values. However the relation 2 > 1 is an expression which evaluates to TRUE so it has a value 1 in C. Likewise if we have an expression x = 10 this has a value which in this case is 10 the value actually assigned.

6) **Conditional Operators:** the operators ?: is called as conditional operator and it is also called as ternary operator since it operates on three operands.

**Syntax: expression1?expression2:expression3;**

In the above conditional expression, if expression1 is TRUE then returns the value of expression2 otherwise returns the value of expression3.

**For example:-**

```
5>12 ? 11: 12;          // returns 12, since 5 not greater than 12.
```

```
10!=5 ? 4 : 3;          // returns 4, since 10 is not equal to 5.
12>8 ? a : b;           // returns the value of a, since 12 is greater than 8.
```

7) **Bitwise Operators:** These operators performs operations on each individual bits of the data value rather than on usual data value. Hence the name bitwise.

These are special operators that act on char or int variables only.

| | |
|---|---|
| & Bitwise AND | \| Bitwise OR |
| ^ Bitwise XOR | ~ Ones Complement |
| >> Shift Right | << Shift left |

Recall that type char is one byte in size. This means it is made up of 8 distinct bits or binary digits normally designated as illustrated below with Bit0 being the Least Significant Bit (LSB) and Bit 7 being the Most Significant Bit (MSB).

The value represented below is 13 in decimal.

| Bit7(MSB) | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0(LSB) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

An integer on a 16 bit OS is two bytes in size and so Bit15 will be the MSB while on a 32 bit system the integer is four bytes in size with Bit31 as the MSB.

**Bitwise AND( & )**

RULE : If any two bits in the same bit position are 1 then the resultant bit in that position is 1 otherwise it is zero.

For Example :- 1011 0010 (178) & 0011 1111 (63) = 0011 0010 (50)

**Bitwise OR ( | )**

RULE : If any two bits in the same bit position are 0 then the resultant bit in that position is 0 otherwise it is 1.

For Example :- 1011 0010 (178) | 0000 1000 (8) = 1011 1010 (186)

**Bitwise XOR ( ^ )**

RULE : If the bits in corresponding positions are different then the resultant bit is 1 otherwise it is 0.

For Example :- 1011 0010 (178) ^ 0011 1100 (60) = 1000 1110 (142)

**Shift Operators, << and >>**

RULE : These move all bits in the operand left or right side by a specified number of places.

Syntax : variable << numberOfPlaces
variable >> numberOfPlaces

For Example :- 2 << 2 = 8 i.e. 0000 0010 becomes 0000 1000

NB : shift left variable by numberOfPlaces multiplies variable by $2^{numberOfPlaces}$
i.e. resultant value = variable * $2^{numberOfPlaces}$
shift left variable by numberOfPlaces divides variable by $2^{numberOfPlaces}$
i.e. resultant value = variable / $2^{numberOfPlaces}$

**Ones Complement ( ~ )**

RULE : bit 1 is set to 0 and bit 0 is set to 1. i.e. Each bit is negated.

For Example :- 1101 0011 becomes 0010 1100

Note: With all of the above bitwise operators we must work with decimal, octal, or hexadecimal values as binary is not supported directly in C.

The bitwise operators are most commonly used in system level programming where individual bits of an integer will represent certain real life entities which are either on or off,

one or zero. The programmer will need to be able to manipulate individual bits directly in these situations. A mask variable which allows us to ignore certain bit positions and concentrate the operation only on those of specific interest to us is almost always used in these situations. The value given to the mask variable depends on the operator being used and the result required.

For Example :- To clear bit 7 of a char variable.

```
char ch = 89 ;  // any value
char mask = 127 ; // 0111 1111
ch = ch & mask ; // clears bit7 of a variable ch ;
```

For Example :- To set bit 1 of an integer variable.

```
int i = 234 ;  // any value
int mask = 2 ;  // a 1 in bit position 2
i = i | mask ; // sets the bit 1 of variable i
```

8) **Special Operators: the following are called as special operators**

a. **sizeof** operator: used to determine the size, in bytes, of the variable or datatype. The sizeof operator returns the amount of memory, in bytes, associated with a variable or a type.

**Syntax**: sizeof ( expression )

For Example:-

```
int x , size ;
size = sizeof(x) ;        // sizeof returns size of integer variable x
printf("The integer x requires %d bytes on this machine", size);
printf( "Doubles take up %d bytes on this machine", sizeof (double) ) ;
// prints the size of double data type in bytes
```

b. **Type Cast** operator:

Type cast operator is used to convert one type of data to specific type of data in the expression which has the following syntax.

**Syntax**: ( type ) expression

For Example,

if we have an integer x, and we wish to use floating point division in the expression x/2 we might do the following

```
( float ) x  /  2
```

which causes x to be temporarily converted to a floating point value and then implicit casting causes the whole operation to be floating point division.

The same results could be achieved by stating the operation as

```
x  /  2.0
```

which essentially does the same thing but the former is more obvious and descriptive of what is happening.

**c. Comma operator** – used to separate the arguments in function header, variables in declaration, combine more than one statement in a single line, etc.

For Example,

```
int a,b,c;          // comma separates three variables
a=b,c=10,d=100;     // comma separates three statements
```

## Q.15 What is type conversion? Explain different types of conversion.

**Ans:**

Converting one type of data into another type of data is called as type conversion. What happens when we write an expression that involves two different types of data, such as multiplying an integer and a floating point number? To perform these evaluations, one of the types must be converted.

There are two types of Type conversions:

1) Implicit type conversion
2) Explicit type conversion

**Implicit type conversion:**

When the types of the two operands in a binary expression are different, C automatically converts one type to another. This is known as implicit type conversion.

Example:

```
char     c = 'A';
int      i = 1234;
long double    d = 3458.0004;

i = c;          // since i is integer type C automatically converts c='A' as c = 65 and then assigns 65 to i

d = i;          // since d is long double, value of i is converted to 1234.0 and it is assigned to d.

X = d + i;      // X = double + int is converted to X = double + double, since d and i are of two different types(int is promoted to long double type), therefore result of X is 4692.0004(3458.0004 + 1234.0)
```

**Explicit type conversion:**

Rather than let the compiler implicitly convert data, we can convert data from one type to another ourself using **explicit type conversion.** Explicit type conversion uses the unary **cast** operator, which has a precedence of 14.

To cast data from one type to another, we specify the new type in parentheses before the value we want to be converted.

For example,

To convert an integer A to a float, we code the expression as given below

```
(float) A
```

One use of the cast operator is to ensure that the result of a divide is a real number. For example, if we calculated ratio of girls and boys (both are integer values) in a class without a cast operator, then the result would be an integer value. So to force the result in fractional form, we cast calculation as shown below.

```
Ratio = (float) No_of_Girls / No_of_Boys;
Or
Ratio = No_of_Girls / (float) No_of_Boys;
```

## Q.16 What is precedence and associativity rule? Explain with precedence table.

## Ans:

**Precedence Rule**: Precedence is used to determine the order in which operators with **different precedence** in a complex expression are evaluated.

**Associativity Rule**: Associativity is used to determine the order in which operators with the **same precedence** are evaluated in a complex expression.

Precedence is applied before associativity to determine the order in which expressions are evaluated. Associativity is then applied, if necessary.

When several operations are combined into one C expression the compiler has to rely on a strict set of precedence rules to decide which operation will take preference.

**The precedence of C operators is given below.**

| Precedence | Operator | Associativity |
|---|---|---|
| 16 | ( ) [ ] -> .(dot) | left to right |
| 15 | ++ --(postincrement/decrement) | left to right |
| 15 | ++ --(preincrement/decrement) | |
| | ! ~ +(unary) -(unary) * & sizeof | right to left |
| 14 | (type) | right to left |
| 13 | * / % | left to right |
| 12 | + - | left to right |
| 11 | << >> | left to right |
| 10 | < <= > >= | left to right |
| 9 | == != | left to right |
| 8 | & | left to right |
| 7 | ^ | left to right |
| 6 | \| | left to right |
| 5 | && | left to right |
| 4 | \|\| | left to right |
| 3 | ? : | right to left |
| 2 | = += -= *= /= | |
| | %= &= ^= \|= | |
| | <<= >>= | right to left |
| 1 | , | left to right |

**Examples:**

The following is a simple example of precedence:

2 + 3 * 4

This expression is actually two binary expressions, with one addition and one multiplication operator. Addition has a precedence of 12, multiplication has a precedence of 13 from above table. This results in the multiplication being done first, followed by the addition. The result of the complete expression is 14.

The following is a simple example of Associativity:

2 * 3 / 4

This expression is actually two binary expressions, with one multiplication and one division operator. But both multiplication and division has a precedence of 13(same precedence). Hence, apply the associativity rule as given in table above i.e left to right. This results in the multiplication being done first, followed by the division. The result of the complete expression is 1.

## Question Appeared in Previous Year Question Papers

## Q.17 WACP to which takes p,t,r as input and compute the simple interest and display the result.

**Ans:**

```
        /* Program to find the simple interest */
#include<stdio.h>
int main()
{
        float p,t,r,si;
        printf("Enter p, t and r\n");
        scanf("%f%f%f",&p,&t,&r);
        si = (p*t*r)/100;
        printf("Simple interest = %f\n",si);
        return 0;
}
```

## Q.18 What is the value of X in the following code segments? Justify your answers.

i)
```
int a,b;
float x;
a=4;
b=5;
x=b/a;
```

ii)
```
int a,b;
float x;
a=4;
b=5;
x = (float) b/a;
```

**Ans:**

i) x = b/a;
x = 5/4; // x = int/int
x= 1.0 // since 5/4(int/int) results as 1 but x is of type float so 1 is converted to float i.e. 1.0

ii) x = (float) b/a;
x = (float) 5/4;
x = 5.0/4; // 5 is converted to 5.0(float type) because of cast operator
x = 5.0/4.0; // 4 is also converted to 4.0 because of implicit type conversion
x = 1.25 // hence, the result is 1.25

## Q.19 WACP to find diameter, area and perimeter of a circle.

**Ans:**

```
        /* program to find diameter, area and perimeter of a circle */
#include<stdio.h>
int main()
{
        float radius, diameter, area, perimeter;
        printf("Enter the radius\n");
        scanf("%f",&radius);
        diameter = 2 * radius;
        area = 3.142 * radius * radius;
        perimeter = 2 * 3.142 * radius;
        printf("Diameter of circle = %f\n",diameter);
        printf("Area of circle = %f\n",area);
        printf("Perimeter of circle = %f\n",perimeter);
        return 0;
}
```

## Q.20 WACP to find area and perimeter of a rectangle.

**Ans:**

```
        /* program to find area and perimeter of a rectangle */
#include<stdio.h>
int main()
{
        float length, breadth, area, perimeter;
        printf("Enter length and breadth\n");
        scanf("%f%f",&length, &breadth);
        area = length * breadth;
        perimeter = 2 * (length + breadth);
        printf("Area of rectangle = %f\n",area);
        printf("Perimeter of rectangle = %f\n",perimeter);
        return 0;
}
```

## Q.21 WACP to find sum and average of three given numbers.

**Ans:**

```
        /* program to find sum and average of three numbers */
#include<stdio.h>
int main()
{
        int a,b,c,sum;
        float avg;
        printf("Enter a, b and c\n");
```

```
        scanf( "%d%d%d" , &a, &b, &c);
        sum = a + b + c;
        avg = sum/3;
        printf("Sum = %d\n",sum);
        printf("Average = %f\n" , avg);
        return 0;
}
```

## Q.22 WACP to convert temperature in Celsius into temperature in Fahrenheit.

**Ans:**

```
        /* program to convert temperature in Celsius into temperature in Fahrenheit */
#include<stdio.h>
int main()
{
        float C,F;
        printf("Enter Temperature in Celsius\n");
        scanf( "%f" , &C);
        F = 1.8 * C + 32;
        printf("Fahrenhiet = %f\n" , F);
        return 0;
}
```

## Q.23 WACP to convert temperature in Fahrenheit into temperature in Celsius.

**Ans:**

```
        /* program to convert temperature in Fahrenheit into temperature in Celsius */
#include<stdio.h>
int main()
{
        float C,F;
        printf("Enter Temperature in Fahrenheit \n");
        scanf( "%f" , &F);
        C = (F – 32)/1.8;
        printf("Celsius = %f\n" , C);
        return 0;
}
```

## Q.24 Write a C program that computes the size of int, float, double and char variables.

**Ans:**

```
/* program to compute size of int, float, double and char variables */
#include<stdio.h>
int main()
{
    int i;
    float f;
    double d;
    char c;
    printf(" Size of integer variable = %d\n" , sizeof(i));
    printf(" Size of float variable = %d\n" , sizeof(f));
    printf(" Size of double variable = %d\n" , sizeof(d));
    printf(" Size of character variable = %d\n" , sizeof(c));
    return 0;
}
```

## Q.25 Write an algorithm to find sum and average of N numbers.

**Ans:**

Algorithm for finding sum and average of N numbers

Step 1: start

Step 2: read the value of N

Step 3: initialize sum as 0

Step 3: until I less than or equal to N

Repeat

sum = sum + I

Done

Step 4: print sum

Step 5: stop

## Q. 26 Write the C expression for the following:

i) A = $\frac{5x+3y}{a+b}$

ii) C = $e^{|x+y-10|}$

iii) D = $\frac{e^{\sqrt{x}}+e^{\sqrt{y}}}{xsin\sqrt{y}}$

iv) B = $\sqrt{s(s-a)(s-b)(s-c)}$

v) E = $x^{25} + y^{35}$

vi) X = $\frac{-b + \sqrt{b^2-4ac}}{2a}$

**Ans:**

i) Given A = $\frac{5x+3y}{a+b}$

Corresponding C expression is A = ( (5*x) + (3*y) ) / (a+b);

ii) Given C = $e^{|x+y-10|}$

Corresponding C expression is C = exp( fabs(x+y-10) );

iii) Given D = $\frac{e^{\sqrt{x}}+e^{\sqrt{y}}}{xsin\sqrt{y}}$

Corresponding C expression is

D = ( exp(sqrt(x)) + exp(sqrt(y)) ) / (x*sin(sqrt(y));

iv) Given B = $\sqrt{s(s-a)(s-b)(s-c)}$

Corresponding C expression is

B = sqrt(s*(s-a)*(s-b)*(s-c));

v) Gievn E = $x^{25} + y^{35}$

Corresponding C expression is

E = pow(x,25) + pow(y,25);

vi) Given X = $\frac{-b + \sqrt{b^2-4ac}}{2a}$

Corresponding C expression is

X = ( -b + sqrt(b*b-4*a*c) ) / (2*a);

**Q. 27 Evaluate the following expression**

**Assume, int i = 10, b=20,c=5,e=2;**

**float q =2.5, d=4.5;**

**i) a/b + (a/(2*b))**

**ii) f = ++a + b-- / q find value of f, a, and b**

**iii) G = b%a++; find value of G and a**

**iv) H = b%++a; find value of H and a**

**v) a+2 < b || !c && a==d || a-2 <=e**

**Ans:**

**i) a/b + (a/(2*b))**

| | |
|---|---|
| 10/20 + ( 10/(2*20)) | // substituite values of variables |
| 10/20 + (10/40) | // inner most parentheses is having highest precedence |
| 10/20 + 0 | //parentheses is having highest precedence so evaluate it |
| | //first, 10/40(int/int) so result is 0 |

0 + 0 // division operators has higher precdence than +,
//10/20(int/int) so result is 0
0 // overall result of expression

ii) **f = ++a + b-- / q find value of f, a, and b**

f = ++10 + 20-- / 2.5 // substiuite values
f = ++10 + 20 / 2.5 // b =19, since postdecrement(--) operator has
//higher precedence evaluate it first according to steps
f = 11 + 20 / 2.5 // a = 11, since preincrement(++) operator has
//higher precedence evaluate it first according to steps
f = 11 + 8.0 // 20/2.5(int/float) so result is floating point
//value(implicit type conversion)
f = 19.0 // 11+8.0(int + float) so final result is also float

**Therefore, f=19.0, a=11, b=19**

iii) **G = b%a++; find value of G and a**

G = 20%10++; // substituite values
G = 20%10 // a=11, postincrement(++) has higher precedence so
// evaluate it first according to steps
G = 0 // % operator results remainder hence answer is 0

**Therefore, G =0, a=11**

iv) **H = b%++a; find value of H and a**

H = 20%++10; // substituite values
H = 20%11 // a=11, preincrement(++) has higher precedence so
// evaluate it first according to steps
H = 9 // % operator results remainder hence answer is 9

**Therefore, H = 9, a = 11**

v) **a+2 < b || !c && a==d || a-2 <=e**

10 + 2 < 20 || **!5** && 10 == 4.5 || 10-2 <= 2 //substituite values

**/* Since unary ! operator is having highest precedence so !5 => !true => false(0) */**

**10 + 2** < 20 || 0 && 10 == 4.5 || 10-2 <= 2

**/* + and – operator have higher precedence and both have same precedence, so use associativity rule i.e. left to right, hence + is evaluted first */**

12 < 20 || 0 && 10==4.5 || **10-2** <= 2

**/* - has higher precedence so evaluate it first,10-2 is 8 */**

**12<20** || 0 && 10 ==4.5 || 8 <=2

**/* < and <= have higher precedence and both have same precedence, so use associativity rule i.e. left to right, hence evaluate < first, 12<20 is true so result is 1 */**

1|| 0 && 10==4.5 || **8<=2**

**/* <= has higher precedence so evaluate it first, 8<=2 is false so result is 0 */**

1 || 0 && **10==4.5** || 0

**/*== has higher precedence so evaluate it first,10==4.5 is false, so result is 0*/**

1 || **0 && 0** || 0

**/* && has higher precedence so evaluate it first, 0&&0 is 0 */**

**1 || 0** || 0

**// use associativity rule of || operator i.e. left to right and evaluate 1||0 is 1 */**

**1||0**

**1 // result of complete expression**

***** ALL THE BEST*****