1. Introduction

Q.4.1.1.1 Explain the usage of header files.
Answer: In C programs, we use many library functions. All the library functions in C have been grouped into different categories. For example, all string processing functions are kept in *string.h* header file. If a program uses any string processing operation, it must include the *string.h* header file. So, instead of including all library header files in a particular C program, we include only those header files, from where the functions have been refered to. So, this approach reduces the size of the object program.

Q.4.1.1.2 Are the following lines executable statements?
include <stdio.h>
define TRUE 1
Answer: Line number 1 is not an executable statement. It is a directive to preprocessor. It directs the preprocessor to replace line 1 by all the functions available in the header file *stdio.h*.
Line number 2 is also not an executable statement. It is a directive to preprocessor. It directs the preprocessor to replace the word TRUE by 1 at every occurrence of the word TRUE in the given C program.

Q.4.1.1.3 Name a computer program which never stops.
Answer: Operating system is a complex computer program that never stops, even when the computer remains idle.

Q.4.1.1.4 What are escape sequences? How do newline and carriage return characters differ?
Answer: C uses certain combination of characters to represent another character. For example, \a and \b represent audible bell and backspace character respectively.
The newline character resets the cursor to the leftmost column on the screen and moves the cursor down one line. But carriage return character resets the cursor to the leftmost column on the current line without advancing to the next line. Consider the following lines of code.

```
printf("Akash\n");
printf("Nandita\n");
```

```
printf("Ava");
printf("\rYeshwant");
```

It generates the following output.
Akash
Nandita
Yeshwant
It prints also the string "Ava" at the third line. But, due to the fourth *printf* statement, the cursor is brought at the leftmost column, and displays string "Yeshwant". Then the string "Ava" gets erased by the string "Yeshwant".

Q.4.1.1.5 What is the difference between the following two statements.

```
printf("\n \n \n");
printf("\n", "\n", "\n");
```

Answer: In a *printf* statement, it is possible to have only one control string.
For the first statement, three blank lines will be produced.
For the second statement, there is a warning. Only one blank line will be produced. The second and third control strings of the second *printf* statement will be ignored.

Q.4.1.1.6 Express 319×10^9 using C language.
Answer: There are many ways one could represent 319×10^9. Some representations are given below.
319.0e9
.319e+12
3.19E11
319E9

Q.4.1.1.7 Explain the following constants using C language.
1000000UL, 071U, 0x12UL, 0xab
Answer: 11000000UL → The number (1000000) in decimal system and it is unsigned long.
071U → The number (71) in octal system and it is unsigned.
0x12UL → The number (12) in hexadecimal system and it is unsigned long.
0xab → The number (ab) in hexadecimal system.

Q.4.1.1.8 What is difference between a character and the corresponding single character string?
Answer: The character constant and the corresponding single character string are not the same. A single-character string consists of two characters - the specified character followed by the null character. The null character is represented by \0.

Q.4.1.1.9 Let variables x and y be of integer and real type respectively. Express statement $y = \frac{1}{1-\frac{1}{1-\frac{1}{1-x}}}$ in C.
Answer: The assignment operation is expressed in C as follows:

```
y = (float)1 / (1 - (float)1 / (1 - (float)1 /(1-x)));
```

To perform the division operation correctly, we need to typecast one of two operands of division operation into $float$ type. Here, integer number 1 gets converted into real number 1.0 by $(float)1$. This process is known as *typecasting*. When $x = 5$, y receives value as 5.0.

Q.4.1.1.10 Discuss priorities of basic arithmetic operations.
Answer: Basic operators and their priorities are given below.

Priority	Operators
high	*, /, % (remainder operator)
medium	+, -
low	=

Q.4.1.1.11 What is associativity rule? Give an example.
Answer: Associativity rule defines the order of execution of consecutive operations within the same precedence group, when there are no parentheses.
For example, $x/y * z\%w$ can be written as $((x/y) * z)\%w$ because, /, *, % have the same priority and their associativity is kept fixed from left to right. Let $x = 4$, $y = 2$, $z = 2$, $w = 3$.
Now, $((x/y) * z)\%w = ((4/2) * 2)\%3 = (2 * 2)\%3 = 1$.
If we evaluate the expression $4/2 * 2\%3$ using a C program, it results 1. But, any arbitrary order of computation would generate wrong result. Since, /, * and % have the same priority, computing % operation first would produce the result 4, since $4/2 * (2\%3) = 4/2 * 2 = 2 * 2 = 4$. Therefore, the associativity plays an important role in computation of an expression.

2. Variables, Operators and Expressions

Q.4.1.2.1 Determine the output of the following statements.

```
p = q = r = 1;
printf("\n p = %d q = %d r = %d", p, q, r);
p = p-(-q---r);
printf("\n p = %d q = %d r = %d", p, q, r);
```

Answer: The output of the above statements are given below.
$p = 1$ $q = 1$ $r = 1$
$p = 3$ $q = 0$ $r = 1$
Consider the statement $p = p - (-q - - - r)$;
The unary $-$ before q has the highest priority in right side of $=$ statement. Again unary $--$ cannot be associated with r as predecrement operator because of presence of binary $-$. So $--$ is treated post decrement operation and must be associated with $-q$. Post decrement takes place only after completion of assignment operation. So, the output is calculated as follows.
$p = 1 - (-1 - 1) = 3$
$q = 1 - 1 = 0$
$r = 1$, as r is not affected.

Q.4.1.2.2 Find the value of the expression

```
e = 7*6/8/3*5*2/4;
```

Answer: The expression is evaluated from left to right. A multiplication is carried out before a division operation, if multiplication comes left of division, otherwise, the division is carried out before multiplication. Thus, $e = 7*6/8/3*5*2/4 = 42/8/3*5*2/4 = 5/3*5*2/4 = 1*5*2/4 = 5*2/4 = 10/4 = 2$. Note that, a division operation results an integer if both the operands are integer.

Q.4.1.2.3 If a computer uses b-bits space to store (i) integer, (ii) unsigned integer then what range of values may fall in this space?
Answer: (i) -2^{b-1} to $+2^{b-1}$ - 1, (ii) 0 to 2^b - 1

Q.4.1.2.4 Determine the output generated by the following statements.

```
i = 0; /* line 1 */
printf ("i = %d \n", i); /* line 2 */
printf ("i = %d \n", i++); /* line 3 */
printf ("i = %d \n", ++i); /* line 4 */
```

Answer: Output generated by the above code is shown below.
$i = 0$
$i = 0$
$i = 2$
Note that, at line 3, the value of i is incremented by 1 after printing the value of i. So, before executing line 4, the value of i is 1. At line 4, the value of i is incremented by 1 before printing the value of i.

Q.4.1.2.5 How many temporary variables are required to exchange the contents of u and v?
Answer: Minimum number of temporary variables required to exchange the content of u and v is 0. The exchange operation can be done using the following codes.
$u = u + v;\ v = u - v;\ u = u - v;$
Let us check the exchange operation. Let $u = 15,\ v = 10$
Then $u = u + v = 15 + 10 = 25$
$v = u - v = 25 - 10 = 15$
$u = u - v = 25 - 15 = 10$

Q.4.1.2.6 Compute a/b and $a\%b$ where, (a, b) = (13, 4), (-13, 4), (13, -4), (-13, -4).
Answer: We have $dividend = divisor \times quotient + remainder$ (1.1)
In C language, / represents integer division operation, and % represents remainder operation, when both the operands are integer. If dividend and division are opposite in sign then quotient is negative and we get the following.

divisor	dividend	quotient
4	13	3
4	-13	-3
-4	13	-3
-4	-13	3

Now using equation (1.1) we obtain the remainder for each case.

divisor	dividend	quotient	remainder
4	13	3	1
4	-13	-3	-1
-4	13	-3	1
-4	-13	3	-1

Q.4.1.2.7 Find the output of the following program segment.

```
float i = 12.870, j = 34.13;
printf("\n Sum of i+j = %f ", i+j);
printf("\n Sum of i+j = %1.2f ", i+j);
printf("\n Sum of i+j = %5.5f ", i+j);
printf("\n Sum of i+j = %g", i+j);
printf("\n Sum of i+j = %e", i+j);
```

Answer: The output of the above statements are given below.
Sum of $i + j = 47.000000$
Sum of $i + j = 47.00$
Sum of $i + j = 47.00000$
Sum of $i + j = 47$
Sum of $i + j = 4.700000e + 01$
Conversion character sequence $\%1.2f$ allows a real number to be displayed with two digits after decimal point. If we use conversion character g, the trailing zeroes, trailing decimal point will not be displayed. By using conversion character e, a floating point value is displayed with an exponent.

Q.4.1.2.8 Find the output of the following code segment.

```
int i = 97;
char c = 'a';
printf("i = %c", i);
printf("\nc = %d", c);
```

Answer: The output of the above code is given below.
$i = a$
$c = 97$
If we display an integer as a character then the integer is treated as an ASCII value and the corresponding character gets printed and vice versa.

Q.4.1.2.9 i is an integer variable. Find the value of i, when

3. Control Statements

Q.4.1.3.1 Find the value of b after executing the following lines of code.

```
int b, a = 15;
b = (a > 5 ? (a <= 10 ? 100 : (a < 15 ? 200 : 300)): 400);
```

Answer: Initially, $a = 15$, and hence $a > 5$ is true. Thus, the expression $a <= 10 \ ? \ 100 : (a < 15 \ ? \ 200 : 300)$ gets evaluated. The condition $a < 15$ becomes false. Thus, the value of righthand side of the assignment statement becomes 300, and it gets assigned to b. Hence, $b = 300$.

Q.4.1.3.2 Express the code given in Q.4.1.3.1 using *if-else* statement.
Answer: An equivalent code using *if-else* statement is given below.

```
int b, a = 15;
if (a > 5)
   if (a <= 10) b = 100;
   else if (a < 15) b = 200;
        else b = 300;
else b = 400;
```

Q.4.1.3.3 Express the following condition without using NOT operator.

```
if (!true)
```

Answer: if (true == 0)

Q.4.1.3.4 Compare *while* loop with *do-while* loop.
Answer: In a *while* loop, the given condition is tested at the beginning. Therefore, the body of the *while* loop may not get executed at all. But, in a *do-while* loop, the given condition is tested at the last. So, the body of *do-while* loop gets executed atleast once. The syntax of a *while* loop in C language is below:

```
while (condition) {
    statement(s);
}
```

The syntax of a *do-while* loop in C language is below:

```
do {
  statement(s);
} while (condition);
```

See flowcharts in Fig. 3.1.

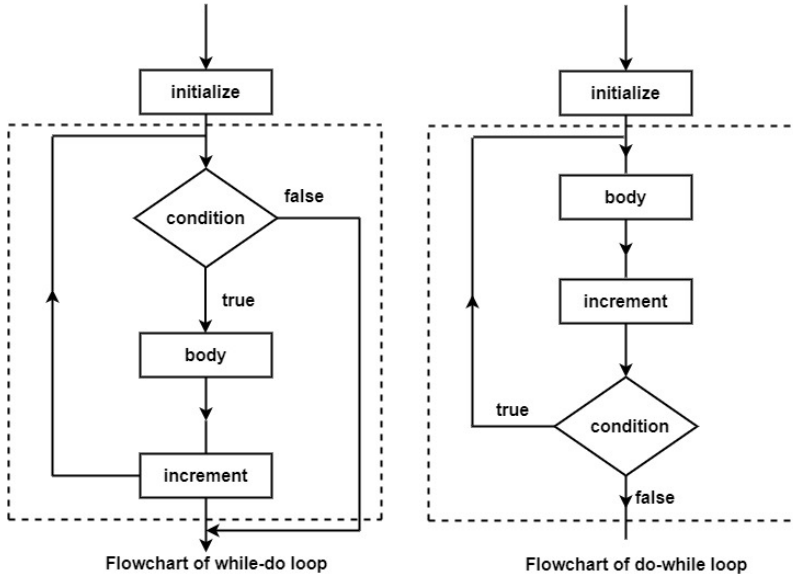

Figure 3.1: Flowcharts of *while* and *do-while* loops (within dotted part)

Q.4.1.3.5 Find the output for the following code.

```
int i = 0, j = 1;
if (i == 0) (j > 1 ? printf("\n123") :  printf("\n ABC"));
else  printf("\n123  ABC");
```

Answer: Condition $i == 0$ is true. Thus, the statement

```
(j > 1 ? printf("\n123") :  printf("\n ABC"));
```

is evaluated. This statement employs conditional or ternary operator (?:). This is similar to *if-else* statement. Here, the condition $j > 1$ is false. So. the statement

```
printf("\n ABC")
```

is executed. The output is given below.

ABC

Q.4.1.3.6 Find the output for the following code.

```
int i = 10;
if (i == 20);
printf ("The value of i is 20");
```

Answer: The output of the above code is given as follows:

```
The value of i is 20
```

Our intension is to display the string only when i is 20. But, the variable i has been initialized to 10. So, the condition ($i == 20$) becomes false. Thus, the body of *if-block* cannot be executed. Note that body of *if-block* contains an empty statement, i.e., only a semicolon (;). The *printf* statement remains outside the body of *if-statement*, and thus, it gets executed irrespective of the condition placed in the *if-statement*.

Q.4.1.3.7 Find the output for the following program segment.

```
a = 10; b = 20;
if (a = b)
  printf ("\n a and b are equal");
else
  printf ("\n a and b are not equal");
```

Answer: The output of the above lines of code is given below:

```
 a and b are equal
```

Note the condition part of if-statement. In the condition part, we have used an assignment statement operation rather than equality condition. As a result, the value of b is assigned to a, and the value of condition becomes 20. The value of condition becomes true, as 20 is non-zero.

Q.4.1.3.8 Find the outputs of Code (a) and Code (b)

```
----------------- Code (a) -----------------
avg = 35.0;
if (avg < 40.0) printf ("Fail");
```