Improvement Test									
Sub:	**PROGRAMMING IN C AND DATA STRUCTURES**							Code:	**15PCD23**
Date:	30 / 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	II	Branch:	**B,D Sec**
Answer Any FIVE FULL Questions									

		Marks	OBE	
			CO	RBT
1(a)	What are the rules for writing an identifier in C?	[04]	CO1	L1
1(b)	List any six constant literals in C with examples.	[06]	CO1	L1

1(a) What are the rules for writing an identifier in C? [04]

Rules for an Identifier

1. An Indetifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
2. The first character of an identifier can only contain alphabet(a-z , A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example *name* and *Name* are two different identifier in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

1(b) List any six constant literals in C with examples. [06]

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals −

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals −

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character Constants

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C that represent special meaning when preceded by a backslash for example, newline (\n) or tab (\t).

Here, you have a list of such escape sequence codes −

Following is the example to show a few escape sequence characters −

```
#include <stdio.h>

int main() {

   printf("Hello\tWorld\n\n");

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Hello World
```

String Literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using white spaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C to define constants –

- Using **#define** preprocessor.
- Using **const** keyword.

The #define Preprocessor

Given below is the form to use #define preprocessor to define a constant –

```
#define identifier value
```

The following example explains it in detail –

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {

   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of area : 50
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows –

```
const type variable = value;
```

The following example explains it in detail –

```
#include <stdio.h>

int main() {

   const int  LENGTH = 10;
   const int  WIDTH = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of area : 50
```

Note that it is a good programming practice to define constants in CAPITALS.

2(a)	Describe relational, logical and bitwise operators with example (it must include the number of operands, operand type and return value).	[06]	CO1	L2

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then

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

True is numeric 1 and false is numeric 0.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

2(b) Describe ***printf()*** and ***scanf()*** with proper syntax and example. [04] CO1 L2

Description

The C library function **int printf(const char *format, ...)** sends formatted output to stdout.

Declaration

Following is the declaration for printf() function.

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

Parameters

- **format** − This is the string that contains the text to be written to stdout. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained below −

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address

n	Nothing printed
%	Character

flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Return Value

If successful, the total number of characters written is returned. On failure, a negative number is returned.

Example

The following example shows the usage of printf() function.

```
#include <stdio.h>

int main ()
{
   int ch;

   for( ch = 75 ; ch <= 100; ch++ )
   {
      printf("ASCII value = %d, Character = %c\n", ch , ch );
   }

   return(0);
}
```

Let us compile and run the above program to produce the following result −

```
ASCII value = 75, Character = K
ASCII value = 76, Character = L
ASCII value = 77, Character = M
ASCII value = 78, Character = N
ASCII value = 79, Character = O
ASCII value = 80, Character = P
ASCII value = 81, Character = Q
ASCII value = 82, Character = R
ASCII value = 83, Character = S
ASCII value = 84, Character = T
ASCII value = 85, Character = U
ASCII value = 86, Character = V
ASCII value = 87, Character = W
ASCII value = 88, Character = X
ASCII value = 89, Character = Y
ASCII value = 90, Character = Z
ASCII value = 91, Character = [
ASCII value = 92, Character = \
ASCII value = 93, Character = ]
ASCII value = 94, Character = ^
ASCII value = 95, Character = _
ASCII value = 96, Character = `
ASCII value = 97, Character = a
ASCII value = 98, Character = b
ASCII value = 99, Character = c
ASCII value = 100, Character = d
```

Description

The C library function ***int fscanf(FILE *stream, const char *format, ...)*** reads formatted input from a stream.

Declaration

Following is the declaration for fscanf() function.

```
int fscanf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **format** − This is the C string that contains one or more of the following items − *Whitespace character, Non-whitespace character* and *Format specifiers*. A format specifier will be as **[=%[*][width][modifiers]type=]**, which is explained below −

argument	**Description**
*	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
width	This specifies the maximum number of characters to be read in the current reading operation.
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f

	and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

fscanf type specifiers

type	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

- **additional arguments** -- Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Example

The following example shows the usage of fscanf() function.

```
#include <stdio.h>
#include <stdlib.h>


int main()
{
   char str1[10], str2[10], str3[10];
   int year;
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fputs("We are in 2012", fp);

   rewind(fp);
   fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

   printf("Read String1 |%s|\n", str1 );
   printf("Read String2 |%s|\n", str2 );
   printf("Read String3 |%s|\n", str3 );
   printf("Read Integer |%d|\n", year );

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2012|
```

3(a) Describe ***typecast, sizeof*** operators with example. What is ***sizeof(void)***? [05] CO1 L2

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows −

```
(type_name) expression
```

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation −

```
#include <stdio.h>

main() {

   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );

}
```

When the above code is compiled and executed, it produces the following result −

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer −

```
#include <stdio.h>

main() {

   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   int sum;

   sum = i + c;
   printf("Value of sum : %d\n", sum );

}
```

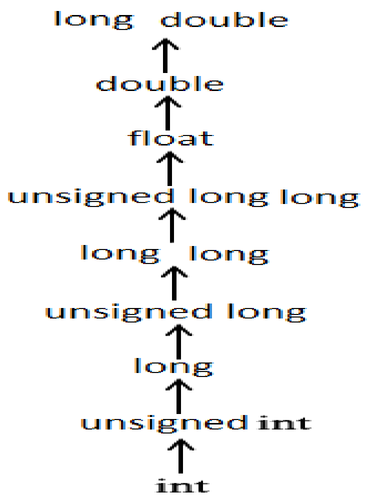
When the above code is compiled and executed, it produces the following result −

```
Value of sum : 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

Usual Arithmetic Conversion

The **usual arithmetic conversions** are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy −



Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression	If Condition is true ? then value X : otherwise value Y

Example

Try following example to understand all the miscellaneous operators available in C −

```
#include <stdio.h>

main() {

   int a = 4;
   short b;
   double c;
   int* ptr;

   /* example of sizeof operator */
   printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
   printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
   printf("Line 3 - Size of variable c= %d\n", sizeof(c) );

   /* example of & and * operators */
   ptr = &a;    /* 'ptr' now contains the address of 'a'*/
   printf("value of a is  %d\n", a);
   printf("*ptr is %d.\n", *ptr);

   /* example of ternary operator */
   a = 10;
   b = (a == 1) ? 20: 30;
   printf( "Value of b is %d\n", b );

   b = (a == 10) ? 20: 30;
   printf( "Value of b is %d\n", b );
}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Size of variable a = 4
Line 2 - Size of variable b = 2
Line 3 - Size of variable c= 8
value of a is  4
*ptr is 4.
```

```
Value of b is 30
Value of b is 20
```

The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && **and ||.**
Let us take the following example to understand the concept

```
#include <stdio.h>

main() {

   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   float sum;

   sum = i + c;
   printf("Value of sum : %f\n", sum );

}
```

When the above code is compiled and executed, it produces the following result −

```
Value of sum : 116.000000
```

Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

3(b) Write a C program to calculate the area of a rectangle. The sides of the rectangle must be taken as input.	[05]	CO3	L3

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int length, breadth, area;
        printf("\nEnter the Length of Rectangle : ");
        scanf("%d", &length);
        printf("\nEnter the Breadth of Rectangle : ");
        scanf("%d", &breadth);
        area = length * breadth;
        printf("\nArea of Rectangle : %d", area);
        return (0);
}
```


4(a) Describe how to declare and initialize a structure element with example. [04] CO1 L2

Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated.

For the above structure of a person, variable can be declared as:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};

int main()
{
    struct person person1, person2, person3[20];
    return 0;
}
```

Another way of creating a structure variable is:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, person3[20];
```

In both cases, two variables *person1*, *person2* and an array *person3* having 20 elements of type **struct person** are created.

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->) (is discussed in structure and pointers tutorial)

Any member of a structure can be accessed as:

```
structure_variable_name.member_name
```

Suppose, we want to access salary for variable *person2*. Then, it can be accessed as:

```
person2.salary
```

Example of structure

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)

```
#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
} dist1, dist2, sum;

int main()
{
    printf("1st distance\n");

    // Input of feet for structure variable dist1
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    // Input of inch for structure variable dist1
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);

    printf("2nd distance\n");

    // Input of feet for structure variable dist2
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);

    // Input of feet for structure variable dist2
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);

    sum.feet = dist1.feet + dist2.feet;
    sum.inch = dist1.inch + dist2.inch;

    if (sum.inch > 12)
    {
        //If inch is greater than 12, changing it to feet.
        ++sum.feet;
        sum.inch = sum.inch - 12;
    }

    // printing sum of distance dist1 and dist2
    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

4(b) Describe structure pointer with example. Describe with example how to pass address of a structure. [06] CO2, CO3 L3

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable −

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

```
struct_pointer->title;
```

Let us re-write the above example using structure pointer.

```
#include <stdio.h>
#include <string.h>

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );

   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );

   return 0;
}

void printBook( struct Books *book ) {

   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

5(a) Describe **fopen(… , …)** with the following different modes***: r, w, a*** (Please mention what will happen in case the file does not exist). [06] CO1 L2

fopen

```
FILE * fopen ( const char * filename, const char * mode );
```

Opens the file whose name is specified in the parameter *filename* and associates it with a stream that can be identified in future operations by the FILE pointer returned.

The operations that are allowed on the stream and how these are performed are defined by the *mode* parameter.

The returned stream is *fully buffered* by default if it is known to not refer to an interactive device (see setbuf).

The returned pointer can be disassociated from the file by calling fclose or freopen. All opened files are automatically closed on normal program termination.

The running environment supports at least FOPEN_MAX files open simultaneously.

Parameters

filename

C string containing the name of the file to be opened.
Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

mode

C string containing a file access mode. It can be:

"r"	**read:** Open file for input operations. The file must exist.
"w"	**write:** Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
"a"	**append:** Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.

"r+"	**read/update:** Open a file for update (both for input and output). The file must exist.
"w+"	**write/update:** Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	**append/update:** Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a `"b"` character has to be included in the *mode* string. This additional `"b"` character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the `"+"` sign for the mixed modes ("rb+", "wb+", "ab+").

The new C standard (C2011, which is not part of C++) adds a new standard subspecifier (`"x"`), that can be appended to any `"w"` specifier (to form `"wx"`, `"wbx"`, `"w+x"` or `"w+bx"/"wb+x"`). This subspecifier forces the function to fail if the file exists, instead of overwriting it.

If additional characters follow the sequence, the behavior depends on the library implementation: some implementations may ignore additional characters so that for example an additional `"t"` (sometimes used to explicitly state a *text file*) is accepted.

On some library implementations, opening or creating a text file with update mode may treat the stream instead as a binary file.

Text files are files containing sequences of lines of text. Depending on the environment where the application runs, some special character conversion may occur in input/output operations in *text mode* to adapt them to a system-specific text file format. Although on some environments no conversions occur and both *text files* and *binary files* are treated the same way, using the appropriate mode improves portability.

For files open for update (those which include a `"+"` sign), on which both input and output operations are allowed, the stream shall be flushed (fflush) or repositioned (fseek, fsetpos, rewind) before a reading operation that follows a writing operation. The stream shall be repositioned (fseek, fsetpos, rewind) before a writing operation that follows a reading operation (whenever that operation did not reach the end-of-file).

Return Value
If the file is successfully opened, the function returns a pointer to a FILE object that can be used to identify the stream on future operations.
Otherwise, a null pointer is returned.
On most library implementations, the errno variable is also set to a system-specific error code on failure.

Example
```
/* fopen example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt","w");
  if (pFile!=NULL)
  {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }
  return 0;
}
```

5(b) Describe any two file input functios with proper syntax, arguments and return value. [04] CO1 L2

fscanf
`int fscanf ( FILE * stream, const char * format, ... );`
Read formatted data from stream
Reads data from the *stream* and stores them according to the parameter *format* into the locations pointed by the additional arguments.

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

Parameters

stream

Pointer to a FILE object that identifies the input stream to read data from.

format

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- **Whitespace character:** the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- see isspace). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- **Non-whitespace character, except format specifier (%):** Any character that is not either a whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a `%` character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- **Format specifiers:** A sequence formed by an initial percentage sign (`%`) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations pointed by the additional arguments.

A *format specifier* for `fscanf` follows this prototype:

```
%[*][width][length]specifier
```

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

specifier	**Description**	**Characters extracted**
`i, u`	Integer	Any number of digits, optionally preceded by a sign (`+` or `-`). Decimal digits assumed by default (`0-9`), but a `0` prefix introduces octal digits (`0-7`), and `0x` hexadecimal digits (`0-f`).
`d`	Decimal	Any number of decimal digits (`0-9`), optionally preceded

	integer	by a sign (+ or -).
o	Octal integer	Any number of octal digits (0-7), optionally preceded by a sign (+ or -).
x	Hexadecimal integer	Any number of hexadecimal digits (0-9, a-f, A-F), optionally preceded by 0x or 0X, and all optionally preceded by a sign (+ or -).
f, e, g a	Floating point number	A series of decimal digits, optionally containing a decimal point, optionally preceeded by a sign (+ or -) and optionally followed by the e or E character and a decimal integer (or some of the other sequences supported by strtod). Implementations complying with C99 also support hexadecimal floating-point format when preceded by 0x or 0X.
c	Character	The next character. If a *width* other than 1 is specified, the function reads exactly *width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
s	String of characters	Any number of non-whitespace characters, stopping at the first whitespace character found. A terminating null character is automatically added at the end of the stored sequence.
p	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format %p in fprintf.
[*characters*]	Scanset	Any number of the characters specified between the brackets. A dash (-) that is not the first character may produce non-portable behavior in some library implementations.
[^*characters*]	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.
n	Count	No input is consumed.

		The number of characters read so far from *stream* is stored in the pointed location.
`%`	`%`	A `%` followed by another `%` matches a single `%`.

Except for `n`, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: *asterisk* (`*`), *width* and *length* (in that order), which are optional and follow these specifications:

sub-specifier	**description**
`*`	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of `hh`, `h`, `l`, `ll`, `j`, `z`, `t`, `L` (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a *length* sub-specifier):

	specifiers					
length	**`d i`**	**`u o x`**	**`f e g a`**	**`c s [] [^]`**	**`p`**	**`n`**
(none)	`int*`	`unsigned int*`	`float*`	`char*`	`void**`	`int*`
`hh`	`signed char*`	`unsigned char*`				`signed char*`
`h`	`short int*`	`unsigned short int*`				`short int*`
`l`	`long int*`	`unsigned long int*`	`double*`	`wchar_t*`		`long int*`
`ll`	`long long int*`	`unsigned long long int*`				`long long int*`
`j`	`intmax_t*`	`uintmax_t*`				`intmax_t*`

z	size_t*	size_t*				size_t*
t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*
L			long double*			

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99.

... *(additional arguments)*

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.
There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.
These arguments are expected to be pointers: to store the result of a `fscanf` operation on a regular variable, its name should be preceded by the *reference operator* (`&`) (see example).

Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (feof or ferror). And, if either happens before any data could be successfully read, EOF is returned.

If an encoding error happens interpreting wide characters, the function sets errno to `EILSEQ`.

Example

```
/* fscanf example */
#include <stdio.h>

int main ()
{
  char str [80];
  float f;
  FILE * pFile;

  pFile = fopen ("myfile.txt","w+");
  fprintf (pFile, "%f %s", 3.1416, "PI");
  rewind (pFile);
  fscanf (pFile, "%f", &f);
  fscanf (pFile, "%s", str);
  fclose (pFile);
  printf ("I have read: %f and %s \n",f,str);
  return 0;
}
```

fgetc

`int fgetc ( FILE * stream );`

Get character from stream

Returns the character currently pointed by the internal file position indicator of the specified *stream*. The internal file position indicator is then advanced to the next character.

If the stream is at the end-of-file when called, the function returns EOF and sets the *end-of-file indicator* for the stream (feof).

If a read error occurs, the function returns EOF and sets the *error indicator* for the stream (ferror).

`fgetc` and getc are equivalent, except that getc may be implemented as a macro in some libraries.

Parameters

stream

Pointer to a FILE object that identifies an input stream.

Return Value

On success, the character read is returned (promoted to an `int` value).
The return type is `int` to accommodate for the special value EOF, which indicates failure:
If the position indicator was at the *end-of-file*, the function returns EOF and sets the *eof indicator* (feof) of *stream*.
If some other reading error happens, the function also returns EOF, but sets its *error indicator* (ferror) instead.

Example

```c
/* fgetc example: money counter */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  int c;
  int n = 0;
  pFile=fopen ("myfile.txt","r");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    do {
      c = fgetc (pFile);
      if (c == '$') n++;
    } while (c != EOF);
    fclose (pFile);
    printf ("The file contains %d dollar sign characters ($).\n",n);
  }
  return 0;
}
```

6(a) Describe void pointer and NULL pointer with example. [04] CO1 L2

void pointer in C

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.

```c
int a = 10;
char b = 'x';

void *p = &a;  // void pointer holds address of int 'a'
p = &b; // void pointer holds address of char 'b'
```

Some Interesting Facts:
1) void pointers cannot be dereferenced. For example the following program doesn't compile.

```
#include<stdio.h>
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

Output:

```
Compiler Error: 'void*' is not a pointer-to-object
type
```

2) The C standard doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of void is 1. For example the following program compiles and runs fine in gcc.

```
#include<stdio.h>
int main()
{
    int a[2] = {1, 2};
    void *ptr = &a;
    ptr = ptr + sizeof(int);
    printf("%d", *(int *)ptr);
    return 0;
}
```

Output:

```
2
```

Note that the above program may not work in other compilers.

6(b) Write a C program to calculate the mean and standard deviation of n real numbers using pointer notation. [06] CO2, 3,4 L3

NULL pointer in C

At the very high level, we can think of NULL as null pointer which is used in C for various purposes. Some of the most common use cases for NULL are
a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
b) To check for null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.
c) To pass a null pointer to a function argument when we don't want to pass any valid memory address.

The example of a) is

```
int * pInt = NULL;
```

The example of b) is

```
if(pInt != NULL) /*We could use if(pInt) as well*/
{ /*Some code*/}
else
{ /*Some code*/}
```

The example of c) is

```
int fun(int *ptr)
{
 /*Fun specific stuff is done with ptr here*/
 return 10;
}
fun(NULL);
```

It should be noted that NULL pointer is different from uninitialized and dangling pointer. In a specific program context, all uninitialized or dangling or NULL pointers are invalid but NULL is a specific invalid pointer which is mentioned in C standard and has specific purposes. What we mean is that uninitialized and dangling pointers are invalid but they can point to some memory address that may be accessible though the memory access is unintended.

```
#include <stdio.h>
int main()
{
 int *i, *j;
 int *ii = NULL, *jj = NULL;
 if(i == j)
 {
  printf("This might get printed if both i and j are same by chance.");
 }
 if(ii == jj)
 {
  printf("This is always printed coz ii and jj are same.");
 }
 return 0;
}
```

7(a) Describe any two of these: mallloc(), calloc(), realloc() and free() with proper syntax and examples. [04] CO5 L2

S.N.	Function & Description
1	**void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will be **size**.
2	**void free(void *address);** This function releases a block of memory block specified by address.
3	**void *malloc(int num);** This function allocates an array of **num** bytes and leave them uninitialized.
4	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize**.

Allocating Memory Dynamically

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows −

```
char name[100];
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below example −

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* allocate memory dynamically */
   description = malloc( 200 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else {
      strcpy( description, "Zara ali a DPS student in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
}
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali
```

```
Description: Zara ali a DPS student in class 10th
```

Same program can be written using **calloc();** only thing is you need to replace malloc with calloc as follows −

```
calloc(200, sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of realloc() and free() functions −

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* allocate memory dynamically */
   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else {
      strcpy( description, "Zara ali a DPS student.");
   }
   /* suppose you want to store bigger description */
   description = realloc( description, 100 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else {
      strcat( description, "She is in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* release memory using free() function */
   free(description);
}
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

You can try the above example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.

7(b) Describe stack and queue datastructure with examples and applications. [06] CO6 L2

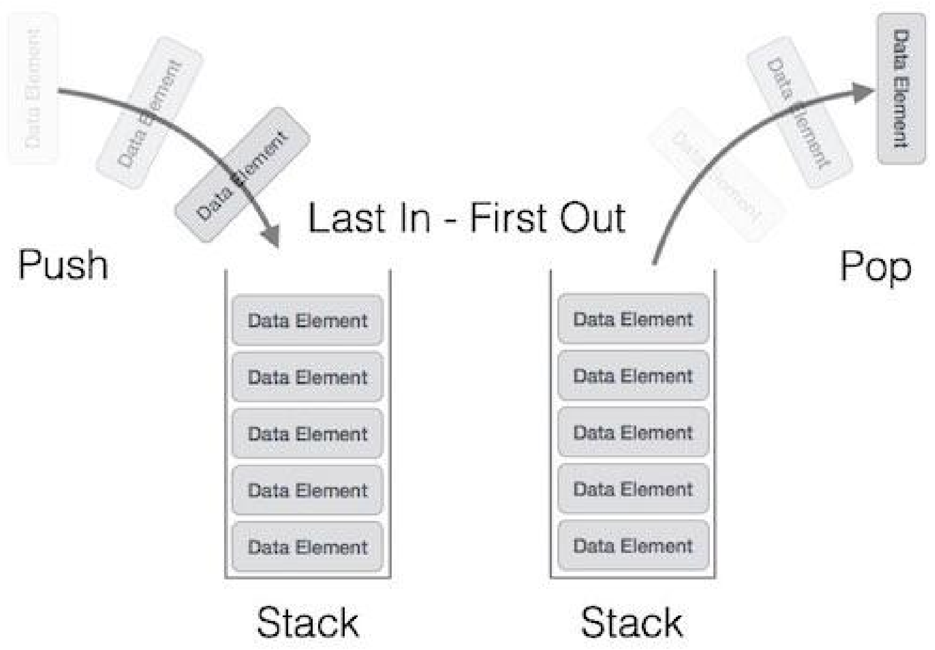
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if top less than 1
      return true
   else
```

```
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

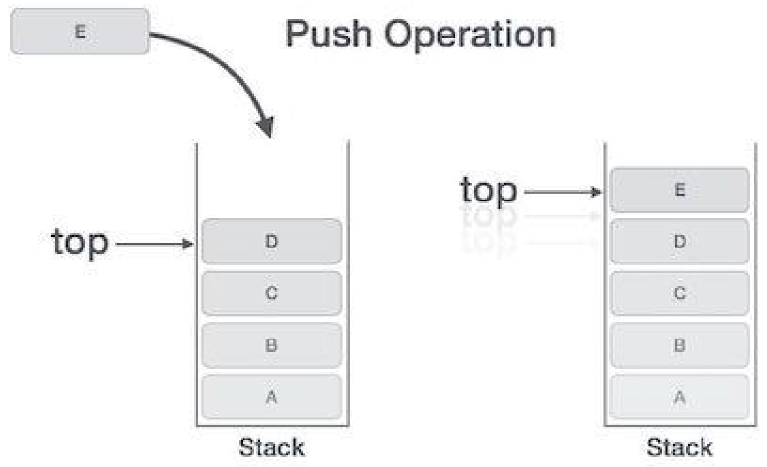
Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

Example

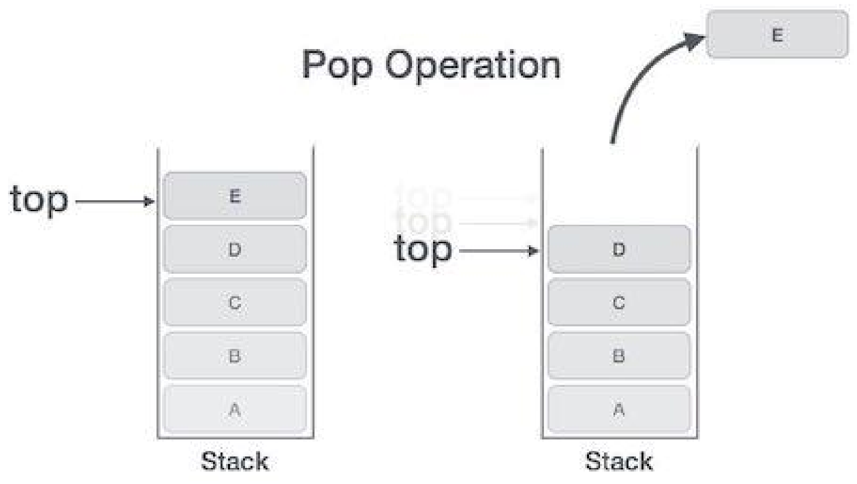
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

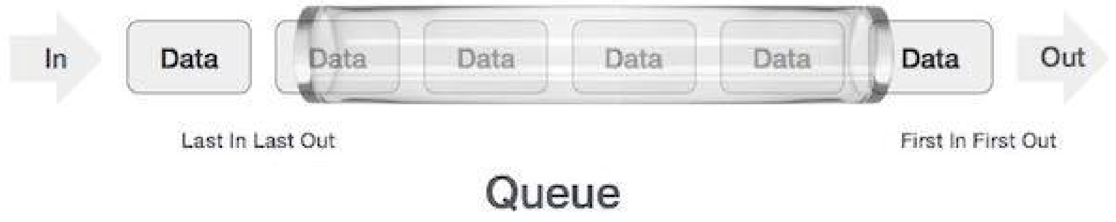
```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.

- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek

   return queue[front]

end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

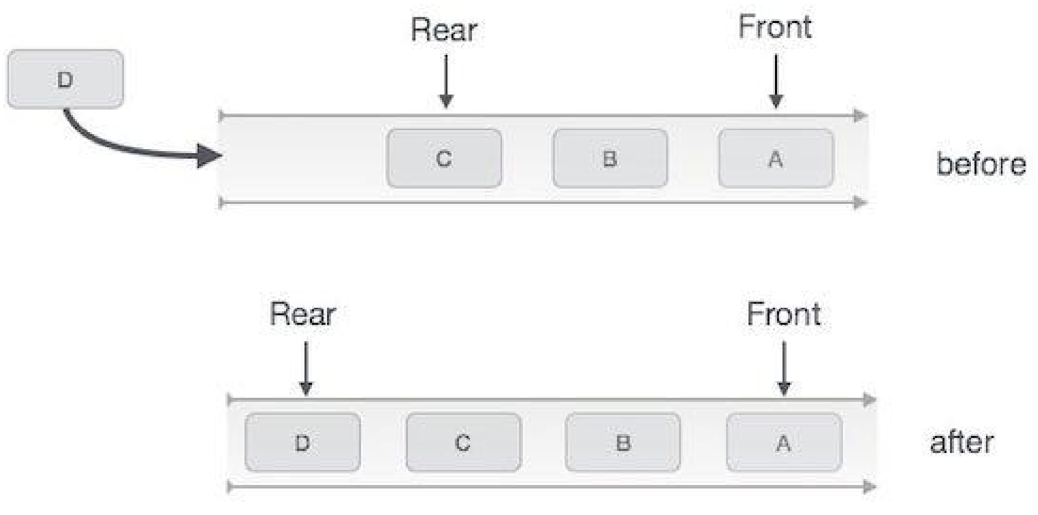
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```

Implementation of enqueue() in C programming language −

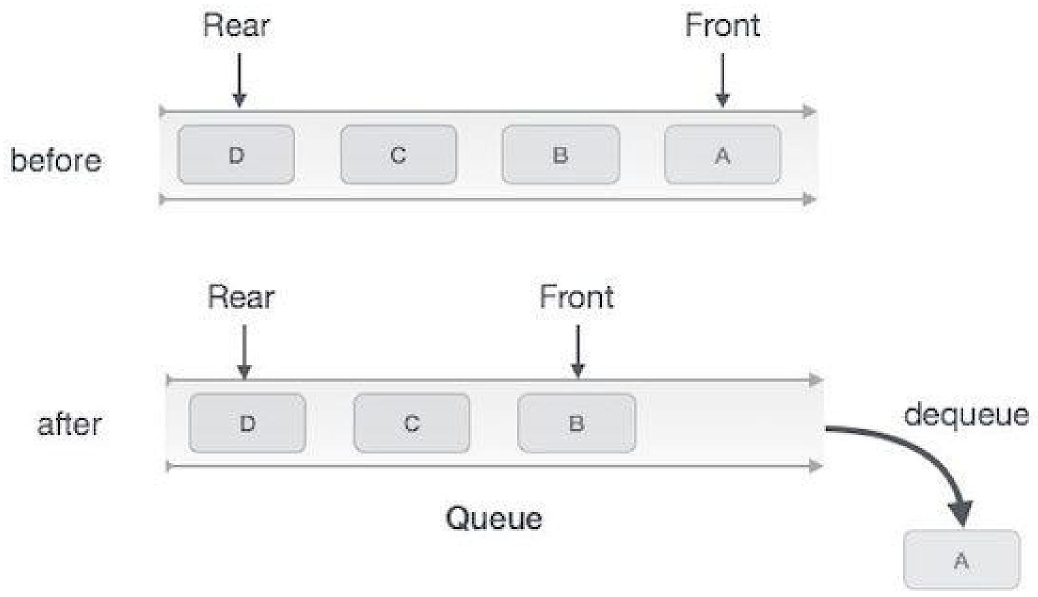
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {
```

[Type text]

```
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

8(a) Write a C program to find the square root of a given number. [06] CO2, CO3 L3

```
#include<stdio.h>
void main()
{
 float m,n;
 float num;
 n=0.0001;   // This is taken small so that we can calculate upto decimal places also
 printf("ENTER A NUMBER : ");
 scanf("%f",&num);

 for(m=0;m<num;m=m+n)
 {
  if((m*m)>num)
  {
   m=m-n;         // This if() is used to calculate the final value as soon as the square of the
number exceeds
   break;      //  the number then we deduct the value exceeded and stop the procedure using break;
this is our final value which is stored in m;
  }
 }
 printf("%.2f",m);
 getch();
 return 1;
}
```

8(b) Describe any two file output functions with proper syntax and example. [04] CO1 L2

fputs ()

Description

The C library function **int fputs(const char *str, FILE *stream)** writes a string to the specified stream up to but not including the null character.

Declaration

Following is the declaration for fputs() function.

```
int fputs(const char *str, FILE *stream)
```

Parameters

- **str** -- This is an array containing the null-terminated sequence of characters to be written.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the string is to be written.

Return Value

This function returns a non-negative value, or else on error it returns EOF.

Example

The following example shows the usage of fputs() function.

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fp = fopen("file.txt", "w+");

   fputs("This is c programming.", fp);
   fputs("This is a system programming language.", fp);

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program, this will create a file **file.txt** with the following content:

```
This is c programming.This is a system programming language.
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

fprintf()

Description

The C library function **int fprintf(FILE *stream, const char *format, ...)** sends formatted output to a stream.

Declaration

Following is the declaration for fprintf() function.

```
int fprintf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **format** − This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained below −

specifier	**Output**
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)

p	Pointer address
n	Nothing printed
%	Character

flags	Description
-	Left-justifies within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is written, a blank space is inserted before the value.
#	Used with o, x or X specifiers. The value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow then no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed after the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is

	specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	**Description**
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **additional arguments** − Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned otherwise, a negative number is returned.

Example

The following example shows the usage of fprintf() function.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);

   fclose(fp);
   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content −

```
We are in 2012
```

Now let's see the content of the above file using the following program −

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
         break;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```