S-18 ACPLD

Q1.a) What is an array? Explain how one dimensional and two-dimensional array are stored in memory. Give example of each.

An **array** is a collection of data items, all of the same type, accessed using a common name. A one-dimensional **array** is like a list; A two dimensional**array** is like a table; The **C** language places no limits on the number of dimensions in an **array**, though specific implementations may

A single-dimensional array is the simplest form of an array that requires only *one* subscript to access an array element. Like an ordinary variable, an array must have been declared before it is used in the program. The syntax for declaring a single-dimensional array is

data_type array_name [size] ;

For example, an array marks of type int and size five can be declared using this statement.

int marks [5];

Initialization of Single- Dimensional Array: Once an array is declared, the next step is to initialize each array element with a valid and appropriate value. Note that unless an array is initialized, all the array elements contain garbage values. An array can be initialized at the time of its declaration. The syntax for initializing an array at the time of its declaration

data_type array_name [size] ={value 1,value 2,…… ,value n};

The values are assigned to the array elements in the order in which they are listed.

That is, value1, value 2 and value n are assigned to the first, second and nth element of the array, respectively.

For example, the array marks can be initialized while declaring using this statement.

int marks[5]={51,62,43,74,55};

If an array is declared and initialized simultaneously, then specifying its size is optional. For example, the statement int marks [] = {51, 62,43,74,55} is also valid.

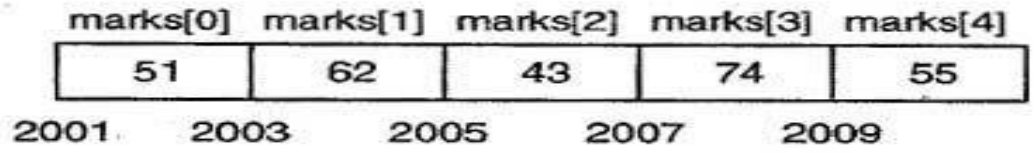
Accessing Single-Dimensional Array Elements: Once an array is declared and initialized, the values stored in the array can be accessed any time. Each individual array element can be accessed using the name of the array and the subscript value. Every element in an array is associated with a unique subscript value, starting from 0 to size-l (where, size refers to the maximum number of elements that can be stored in the array). The syntax for accessing the values stored in a single dimensional array is

array_name [subscript]

For examp1e, the elements of the array marks can be referred to as marks [0], marks [1],

Marks [2], marks [3] and marks [4], respectively.

Single-dimensional arrays are always allocated contiguous blocks of memory. This implies that all the elements in an array are always stored next to each other. The memory representation of the array marks is shown in Figure. As each element is of the type int (that is, 2 bytes long), the array marks occupies ten contiguous bytes in memory and these bytes are reserved in the memory at the compile-time.



Memory Representation of marks

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {
   {10, 11, 12, 13},
   {14, 15, 16, 17}
};
```

OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let's understand this with the help of few examples –

```
/* Valid declaration*/
int abc[2][2] = {1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = {1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[][] = {1, 2, 3 ,4 }
/* Invalid because of the same reason  mentioned above*/
int abc[2][] = {1, 2, 3 ,4 }
```

How to store user input data into 2D array

We can calculate how many elements a two dimensional array can have by using this formula:

The array arr[n1][n2] can have n1*n2 elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has first subscript value as 5 and second subscript value as 4.

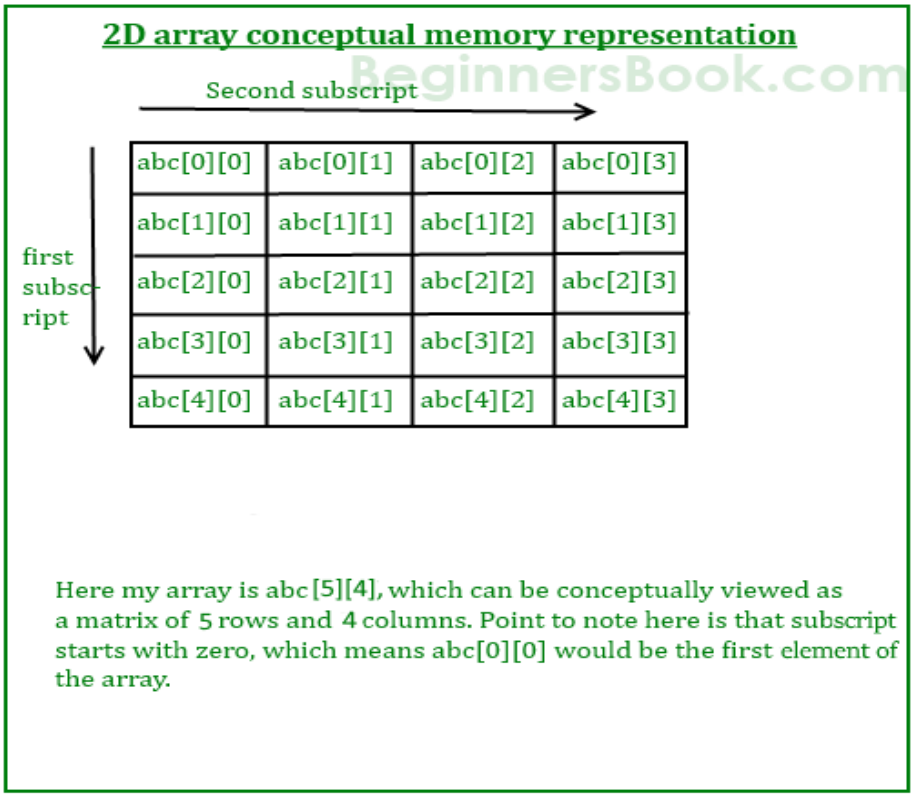
So the array abc[5][4] can have 5*4 = 20 elements.

To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be abc[0][0], abc[0][1], abc[0][2]…so on.

```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int abc[5][4];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<5; i++) {
```

```
    for(j=0;j<4;j++) {
      printf("Enter value for abc[%d][%d]:", i, j);
      scanf("%d", &abc[i][j]);
    }
  }
  return 0;
}
```

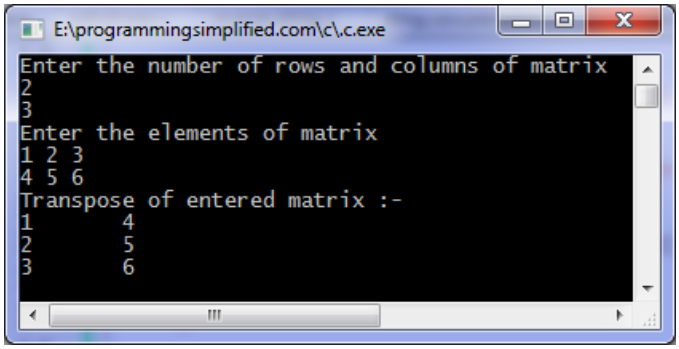
In above example, I have a 2D array abc of integer type. Conceptually you can visualize the above array like this:



b) Write a program to find transpose of a matrix.

```
#include <stdio.h>

int main()
{
  int m, n, c, d, matrix[10][10], transpose[10][10];

  printf("Enter the number of rows and columns of matrix\n");
  scanf("%d%d", &m, &n);

  printf("Enter elements of the matrix\n");

  for (c = 0; c < m; c++)
    for(d = 0; d < n; d++)
      scanf("%d", &matrix[c][d]);
```

```
  for (c = 0; c < m; c++)
    for( d = 0 ; d < n ; d++ )
      transpose[d][c] = matrix[c][d];

  printf("Transpose of the matrix:\n");

  for (c = 0; c < n; c++) {
    for (d = 0; d < m; d++)
      printf("%d\t", transpose[c][d]);
    printf("\n");
  }

  return 0;
}
```



Q2a) What are structures? Give different way to declared them. When does compiler know to reserve space in memory for members of structure s.

A structure is a user defined data type in C. A structure creates a data type that can be used to group items of possibly different types into a single type.

How to create a structure?

'struct' keyword is used to create a structure. Following is an example.

```
struct addrress
{
  char name[50];
  char street[100];
  char city[50];
  char state[20];
  int pin;
};
```

How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.
struct Point
```

```
{
  int x, y;
} p1;  // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
  int x, y;
};

int main()
{
  struct Point p1;  // The variable p1 is declared like a normal variable
}
```

How to initialize structure members?

Structure members cannot be initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
  int x = 0;  // COMPILER ERROR:  cannot initialize members here
  int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members can be initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
  int x, y;
};

int main()
{
  // A valid initialization. member x gets value 0 and y
  // gets value 1.  The order of declaration is followed.
  struct Point p1 = {0, 1};
}
```

How to access structure elements?

Structure members are accessed using dot (.) operator.

```
struct Point
{
  int x, y;
};

int main()
{
  struct Point p1 = {0, 1};
```

```
  // Accesing members of point p1
  p1.x = 20;
  printf ("x = %d, y = %d", p1.x, p1.y);

  return 0;
}
```

b) Explain any three.

i) Enumeration.
ii) Typedef.
iii) Bitfield.
iv) Sizeof.

Enumeration (or enum) in C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

```
enum State {Working = 1, Failed = 0};
```

The keyword 'enum' is used to declare new enumeration types in C and C++. Following is an **example of enum declaration.**

```
// The name of enumeration is "flag" and the constant
// are the values of the flag. By default, the values
// of the constants are as follows:
// constant1 = 0, constant2 = 1, constant3 = 2 and
// so on.
enum flag{constant1, constant2, constant3, ..... };
```

Variables of type enum can also be defined. They can be defined in two ways:

```
// In both of the below cases, "day" is
// defined as the variable of type week.

enum week{Mon, Tue, Wed};
enum week day;

// Or

enum week{Mon, Tue, Wed}day;
```

typedef, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers –

```
typedef unsigned char BYTE;
```

After this type definition, the identifier BYTE can be used as an abbreviation for the type unsigned char, for example..

```
BYTE  b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows −

typedef unsigned char byte;

You can use typedef to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows −

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

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

int main( ) {

   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;

   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);

   return 0;
}
```

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure −

```
struct {

   type [member_name] : width ;

};
```

The following table describes the variable elements of a bit field −

Sr.No.	Element & Description
1	**type** An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	**member_name**

	The name of the bit-field.
3	**width** The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

```
struct {
  unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value.

Sizeof is a much used operator in the C programming language. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of unsigned integral type which is usually denoted by size_t. sizeof can be applied to any data-type, including primitive types such as integer and floating-point types, pointer types, or compound datatypes such as Structure, union etc.

Usage
sizeof() operator is used in different way according to the operand type.

1. **When operand is a Data Type.**

When sizeof() is used with the data types such as int, float, char… etc it simply return amount of memory is allocated to that data types.

```
#include<stdio.h>
int main()
{
   printf("%d\n",sizeof(char));
   printf("%d\n",sizeof(int));
   printf("%d\n",sizeof(float));
   printf("%d", sizeof(double));
   return 0;
}
```

Q.3a) Write a program to copy abc.txt file into xyz.txt file

```
#include <stdio.h>
#include <stdlib.h> // For exit()

int main()
{
   FILE *fptr1, *fptr2;
   char filename[100], c;

   printf("Enter the filename to open for reading \n");
   scanf("%s", filename);

   // Open one file for reading
   fptr1 = fopen(filename, "r");
   if (fptr1 == NULL)
   {
      printf("Cannot open file %s \n", filename);
      exit(0);
   }

   printf("Enter the filename to open for writing \n");
   scanf("%s", filename);

   // Open another file for writing
   fptr2 = fopen(filename, "w");
   if (fptr2 == NULL)
   {
      printf("Cannot open file %s \n", filename);
      exit(0);
   }

   // Read contents from file
   c = fgetc(fptr1);
   while (c != EOF)
   {
      fputc(c, fptr2);
      c = fgetc(fptr1);
   }

   printf("\nContents copied to %s", filename);

   fclose(fptr1);
   fclose(fptr2);
   return 0;
}
```

3b) Explain fopen( ) function in detail with proper example

fopen() function is used for opening a file.
Syntax:

```
FILE pointer_name = fopen ("file_name", "Mode");
```

pointer_name can be anything of your choice.
file_name is the name of the file, which you want to open. Specify the full path here like "C:\\myfiles\\newfile.txt".
While opening a file, you need to specify the mode. The mode that we use to read a file is "r" which is "read only mode".
for example:

```
FILE  *fp;
fp = fopen("C:\\myfiles\\newfile.txt", "r");
```

The address of the first character is stored in pointer fp.
How to check whether the file has opened successfully?
If file does not open successfully then the pointer will be assigned a NULL value, so you can write the logic like this:
This code will check whether the file has opened successfully or not. If the file does not open, this will display an error message to the user.

```
..
FILE fpr;
fpr = fopen("C:\\myfiles\\newfile.txt", "r");
if (fpr == NULL)
{
   puts("Error while opening file");
   exit();
}
```

Various File Opening Modes:
The file is opened using fopen() function, while opening you can use any of the following mode as per the requirement.
Mode "r": It is a read only mode, which means if the file is opened in r mode, it won't allow you to write and modify content of it. When fopen() opens a file successfully then it returns the address of first character of the file, otherwise it returns NULL.
Mode "w": It is a write only mode. The fopen() function creates a new file when the specified file doesn't exist and if it fails to open file then it returns NULL.
Mode "a": Using this mode Content can be appended at the end of an existing file. Like Mode "w", fopen() creates a new file if it file doesn't exist. On unsuccessful open it returns NULL.
File Pointer points to: last character of the file.
Mode "r+": This mode is same as mode "r"; however you can perform various operations on the file opened in this mode. You are allowed to read, write and modify the content of file opened in "r+" mode.
File Pointer points to: First character of the file.
Mode "w+": Same as mode "w" apart from operations, which can be performed; the file can be read, write and modified in this mode.
Mode "a+": Same as mode "a"; you can read and append the data in the file, however content modification is not allowed in this mode.

c) List various error handling function in files

it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(). and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The **strerror()** function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

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

extern int errno ;

int main () {

   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");

   if (pf == NULL) {

      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
```

```
  } else {


    fclose (pf);
  }


  return 0;
}
```

Q.4a) Write program to count number of lines, words present in the file "PQr.txt"

```
#include <stdio.h>

int main()
{
 FILE *fp;
 char filename[100];
 char ch;
 int linecount, wordcount, charcount;

 // Initialize counter variables
 linecount = 0;
 wordcount = 0;
 charcount = 0;

 // Prompt user to enter filename
  printf("PQr.txt :");
  gets(filename);

   // Open file in read-only mode
  fp = fopen(filename,"r");

  // If file opened successfully, then write the string to file
  if ( fp )
  {
        //Repeat until End Of File character is reached.
          while ((ch=getc(fp)) != EOF) {
                // Increment character count if NOT new line or space
```

```
                if (ch != ' ' && ch != '\n') { ++charcount; }

               // Increment word count if new line or space character
                if (ch == ' ' || ch == '\n') { ++wordcount; }

               // Increment line count if new line character
                if (ch == '\n') { ++linecount; }



        }

        if (charcount > 0) {
            ++linecount;
            ++wordcount;
        }
   }
  else
    {
      printf("Failed to open the file\n");
     }

   printf("Lines : %d \n", linecount);
   printf("Words : %d \n", wordcount);
   printf("Characters : %d \n", charcount);

getchar();
return(0);
}
```

b) Explain command line argument with example

The most important function of C is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

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

int main(int argc, char *argv[])
{
   int i;
   if( argc >= 2 )
   {
      printf("The arguments supplied are:\n");
      for(i = 1; i < argc; i++)
      {
         printf("%s\t", argv[i]);
      }
   }
   else
   {
      printf("argument list is empty.\n");
   }
   return 0;
}
```

c) Write following function:

i) ftell( )

ii) ferror( )

iii) fputs( )

iv) fclose( )

.

i)ftell function returns the current file position indicator for the stream pointed to by stream.

Declaration

Following is the declaration for ftell() function.

long int ftell(FILE *stream)

Parameters

stream − This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns the current value of the position indicator. If an error occurs, -1L is returned, and the global variable errno is set to a positive value.

ii) ferror(FILE *stream) tests the error indicator for the given stream.

Declaration
Following is the declaration for ferror() function.

int ferror(FILE *stream)

Parameters
stream − This is the pointer to a FILE object that identifies the stream.

Return Value
If the error indicator associated with the stream was set, the function returns a non-zero value else, it returns a zero value.

iii)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.

iv) fclose(FILE *stream) closes the stream. All buffers are flushed.

Declaration

Following is the declaration for fclose() function.

```
int fclose(FILE *stream)
```

Parameters

- **stream** − This is the pointer to a FILE object that specifies the stream to be closed.

Return Value

This method returns zero if the stream is successfully closed. On failure, EOF is returned.

Q5a) Compare static memory allocation with dynamic memory allocation

Static vs Dynamic Memory Allocation	
Static memory allocation is a method of allocating memory, and once the memory is allocated, it is fixed.	Dynamic memory allocation is a method of allocating memory, and once the memory is allocated, it can be changed.
Modification	
In static memory allocation, it is not possible to resize after initial allocation.	In dynamic memory allocation, the memory can be minimized or maximize accordingly.
Implementation	
Static memory allocation is easy to implement.	Dynamic memory allocation is complex to implement.
Speed	
In static memory, allocation execution is faster than dynamic memory allocation.	In dynamic memory, allocation execution is slower than static memory allocation.
Memory Utilization	
In static memory allocation, cannot reuse the unused memory.	Dynamic memory allocation allows reusing the memory. The programmer can allocate more memory when required . He can release the memory when necessary.

b) Explain calloc( ), malloc( ), realloc( ), and free( ) function with syntax

malloc()

The name malloc stands for "memory allocation".
The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation".
The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

```
ptr = (cast-type*)calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of `n` elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by `ptr`.

Example #1: Using C malloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

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

int main()
{
   int num, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &num);

   ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }

   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
```

```
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Example #2: Using C calloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

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

int main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);

    ptr = (int*) calloc(num, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize.

Example #3: Using realloc()

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

int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);
```

```
    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\t",ptr + i);

    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2 * sizeof(int));
    for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
    return 0;
}
```

6a) What are pointers? Also explain pointer arithmetic and pointers operators.

pointer is a variable that stores the address of another variable. Unlike other variables that hold values of a certain type, **pointer** holds the address of a variable.
Following arithmetic operations are possible on pointer in C language:

Increment
Decrement
Addition
Subtraction
Comparison

Incrementing Pointer in C

Incrementing a pointer is used in array because it is contiguous memory location. Moreover, we know the value of next location.

Increment operation depends on the data type of the pointer variable. The formula of incrementing pointer is given below:

new_address= current_address + i * size_of(data type)

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p);
return 0;
}
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. The formula of decrementing pointer is given below:

new_address= current_address - i * size_of(data type)

```
#include <stdio.h>

void main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p-1;

printf("After decrement: Address of p variable is %u \n",p);

}
```

Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

new_address= current_address + (number * size_of(data type))

```
#include<stdio.h>

int main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p+3;  //adding 3 to pointer variable

printf("After adding 3: Address of p variable is %u \n",p);

return 0;

}
```

Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. The formula of subtracting value from pointer variable is given below:

new_address= current_address - (number * size_of(data type))

```
#include<stdio.h>

int main(){

int number=50;

int *p;//pointer to int

p=&number;//stores the address of number variable

printf("Address of p variable is %u \n",p);

p=p-3; //subtracting 3 from pointer variable

printf("After subtracting 3: Address of p variable is %u \n",p);

return 0;

}
```

b) Write a program to swap two numbers using pointer.

```
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
   int num1 = 5, num2 = 10;

   // address of num1 and num2 is passed to the swap function
   swap( &num1, &num2);
   printf("Number1 = %d\n", num1);
   printf("Number2 = %d", num2);
   return 0;
}
void swap(int * n1, int * n2)
{
   // pointer n1 and n2 points to the address of num1 and num2 respectively
   int temp;
   temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
```

c) Differentiate pointer to structure and structure pointer

7a) What is the difference between graphics mode and text mode.

Text mode	Graphics mode
1)In text mode the content of screen is internally represent in term of character	1)In graphics mode the content of screen is internally represented in terms of pixels
2) The screen consist of a uniform grid of character cell.	**2)** the screen is treated as array of pixels
3) In text mode we can display only text	**3)** In graphics mode we can display anything that can be made using pixels.
4) No need to include special header file for using text mode	**4)** Graphics.h has to be included in the program using graphics mode.
5) In text mode the consumption of memory is lower	**5)** In graphics mode consumption of memory is high.
6) The screen manipulation is faster	6) The screen manipulation is slower

b) Explain initgraph( ) and closegraph( ) with example.

Initialization – initgraph() function

First of all we call the initgraph() function that will initialize the graphics mode on the computer. The method initigraph() has the following prototype.

initgraph(&graphdriver,&graphmode,"pathtodriver");

The method initgraph() initializes the graphics system by loading the graphics driver from disk (or validating a registered driver) then putting the system into graphics mode. The method initgraph() also resets all graphics settings (color, palette, current position, viewport, etc.) to their defaults. The initialization result is set to 0 which can be retrieved by calling graphresult().

The initgraph() method has the following parameters

graphdriver

This is an integer value that specifies the graphics driver to be used. You can give graphdriver a value using a constant of the graphics_drivers enumeration type which is listed in graphics.h . Normally we use value as "0" (requests auto-detect). Other values are 1 to 10 and description of each enumeration type is

graphmode

This is an integer value that specifies the initial graphics mode (unless *graphdriver = DETECT). If*graphdriver = DETECT , then initgraph() method sets *graphmode to the highest resolution available for the detected graphics driver. You can give *graphmode a value using a constant of the graphics_modes enumeration type and description of each enumeration type is

Pathtodriver

Specifies the directory path where initgraph() looks for graphics drivers (*.BGI) first.

1. If they're not there, initgraph() method looks in the current directory.
2. If pathtodriver is null, the driver files must be in the current directory.

Both *graphdriver and *graphmode parameters must be set to valid graphics_drivers and graphics_mode values or you'll get unpredictable results. (The exception is graphdriver = DETECT.)

After a call to initgraph() , *graphdriver is set to the current graphics driver, and *graphmode is set to the current graphics mode. You can tell initgraph to use a particular graphics driver and mode, or to auto detect the attached video adapter at run time and pick the corresponding driver. If you tell initgraph to auto detect, it calls detectgraph to select a graphics driver and mode.
The initgraph() method loads a graphics driver by allocating memory for the driver (through_graphgetmem() method call), then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file.
Here is a simple program that initializes the graphics mode in C programming language and print the line in graphics mode.
.

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

int main(void)
{
  /* request auto detection */
  int gdriver = DETECT, gmode, errorcode;

  /* initialize graphics mode */
  initgraph(&gdriver, &gmode, "");

  /* read result of initialization */
  errorcode = graphresult();

  if (errorcode != grOk) /* an error occurred */
  {
    printf("Graphics error: %s\n", grapherrormsg(errorcode));
    printf("Press any key to halt:");
    getch();
    exit(1); /* return with error code */
  }

  /* draw a line */
  line(0, 0, getmaxx(), getmaxy());

  /* clean up */
  getch();
  closegraph();
  return 0;
}
```

closegraph();

This function unloads the graphics drivers and returns the screen back to text mode.

```
#include<graphics.h>
```

```
#include<stdio.h>


main()

{

 int gd=DETECT,gm;

 int i,x,y;

 initgraph(&gd,&gm,"");

 line(0,0,640,0);

 line(0,0,0,480);

 line(639,0,639,480);

 line(639,479,0,479);

 for(i=0;i<=1000;i++)

 {

  x=rand()%639;

  y=rand()%480;

  putpixel(x,y,15);

 }

 getch();

 closegraph();

}
```

b) Explain video Adapter in detail.

Q8a) Write a menu driven program to draw line, circle, rectangle, ellipse and arc on the screen.

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
 int gm,gd=DETECT,op,x,y,r,lux,luy,rex,rey,xs,ys,xe,ye,dd;
 initgraph(&gd,&gm,"C:\\TC\\BGI");
 do
 {
  printf("\n1.Circle");
  printf("\n2.Line");
  printf("\n3.Rectangle");
```

```
  printf("\n4.Square");
  printf("\n5.Exit");
  printf("\nEnter your choice:");
  scanf("%d",&op);
  switch(op)
  {
   case 1:
    printf("\nEnter the center and the radius of the circle((x,y) and (r)):");
    scanf("%d%d%d",&x,&y,&r);
    circle(x,y,r);
    break;
   case 2:
    printf("\nEnter the start and end coordinates for line((xs,ys) and(xe,ye)):");
    scanf("%d%d%d%d",&xs,&ys,&xe,&ye);
    line(xs,ys,xe,ye);
    break;
   case 3:
    printf("\nEnter the left upper and right bottom coordinates for rectangle((lux,luy)
and(rex,rey)):");
    scanf("%d%d%d%d",&lux,&luy,&rex,&rey);
    rectangle(lux,luy,rex,rey);
    break;
   case 4:
    printf("\nEnter the start coordinates and the length of the edge of a square(lux,luy,dd):");
    scanf("%d%d%d",&lux,&luy,&dd);
    rectangle(lux,luy,lux+dd,luy+dd);
    break;
   case 5:
    exit(0);
   default:
    printf("\nInvalid choice");
    break;
  }
 }while(op!=5);
 getch();
 closegraph();
}
```

b) Write a program to draw five chains of circles with different colors.

9a) Compare recursion and iteration.

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function	Allows the set of instructions to be

BASIS FOR COMPARISON	RECURSION	ITERATION
	calls the function itself.	repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead	No overhead of repeated function

BASIS FOR COMPARISON	RECURSION	ITERATION
	of repeated function calls.	call.
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

b) Define model of computation. List and explain various model of computations.

11a)List and discuss features of object oriented programming.

- Objects
- Classes
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

***Objects*:**

Objects are the basic run-time entities in an object-oriented programming. They may represents a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represents user-defined data such as vector, time and lists. When the program is executed, the object interact by sending message to one another.

***Classes*:**

Objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variable of the type class. Once a class has been defined, we can create a number of objects belonging to that class. A class is a collection of objects of similar type.

Ex. Fruit mango;

***Data Abstraction*:**

Abstractions refer to the act of representing essential features without including background details or explanation. They are commonly known as Abstraction Data Type(ADT).

Encapsulation:

The wrapping up of data and functions into single unit is known as *encapsulation*. Data encapsulation is a striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program.

Inheritance:

Inheritance is the process by which objects of one class acquire the properties of object of another class. The class whose members are inherited is called the Base class and the class that inherits those members is called Derived class. It supports class of hierarchical classification.

The concept of inheritance provides the ideas of reusability. This means we can add essential features to an exciting class without modifying it.

Polymorphism:

Polymorphism is another OOP concept. Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors at different instances.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.