

2066 Shrawan Solution

1.a) What are the demerits of C programming? Flowchart is representation of an algorithm. Justify the statement with suitable example.

Ans: C programming has its merits and demerits, some of those demerits are listed below:

1. It is a case sensitive language hence a small error can cause a large blunder.
2. The lack of constructor and destructor makes it harder to use.
3. No object orientation is present in it making it harder to reuse the code.
4. The run time error makes it a bit tedious to check the error since it only shows the error when running the program.

A flowchart is a set of boxes and shapes connected with lines and arrows to simplify the given problem to be solved. An algorithm is a set of instructions to be followed to carry out the solutions of a problem. As presented above an algorithm is just the same as a flowchart but without the boxes and arrows. Which can be further explained by the representations shown below.

Algorithm for adding two numbers

Step 1: Start

Step 2: Declare n1, n2, sum

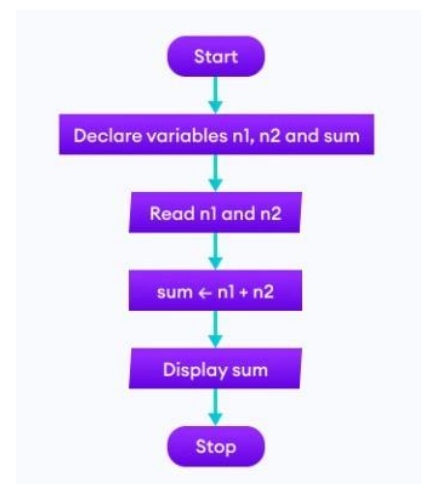
Step 3: Read n1, n2

Step 4: $sum = n1 + n2$

Step 5: Display sum

Step 6: Stop

Flowchart for adding two numbers

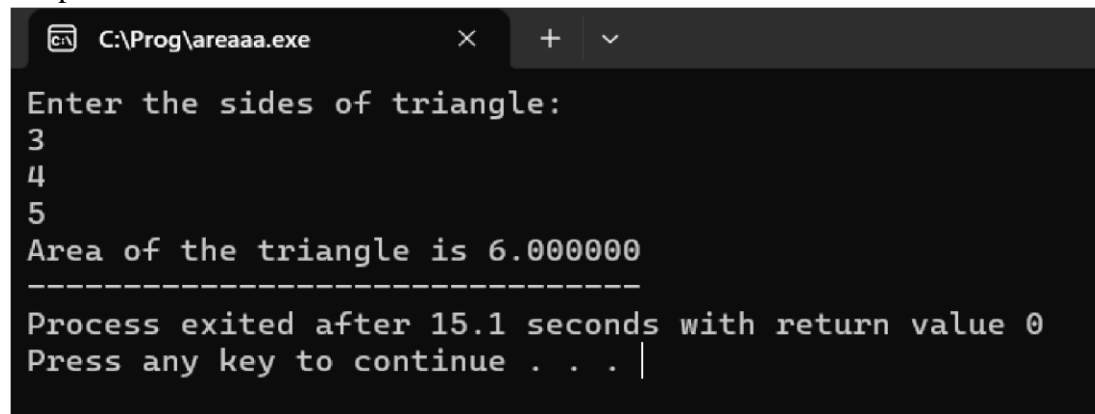


1.b) Write a program to read three sides of a triangle and print area for valid data and to print "Invalid data" if either one side of triangle is greater or equals to the sum of other two sides. [Area= $\sqrt{s(s-a)(s-b)(s-c)}$], where a, b, c are three sides and $s = (a + b + c)/2$.

Ans: To execute the given program we'll need to use conditional statements like else if.

```
#include<stdio.h> #include<math.h> int main ()
{ int a,b,c; float s,brea,area; printf("Enter the
  sides of triangle: \n");
  scanf("%d%d%d",&a,&b,&c); if (a>=(b+c) ||
  b>=(a+c) || c>=(a+b))
  { printf("Invalid Data");
  }
  else
  {
  s=(a+b+c)/2; brea=s*(s-a)
  *(s-b) *(s-c); area=sqrt(brea);
  printf("Area of the triangle is %f",area);
  }
}
```

Output:



```
C:\Prog\areaaa.exe  x  +  v
Enter the sides of triangle:
3
4
5
Area of the triangle is 6.000000
-----
Process exited after 15.1 seconds with return value 0
Press any key to continue . . . |
```

2.a) Describe the meaning of precedence and associativity of operators with suitable example. How much memory is required for long integer and its range?

Ans) Precedence and associativity are concepts in programming that determine the order in which operators are evaluated in an expression.

- a) Precedence: Operator precedence defines the order in which different operators are evaluated when an expression contains multiple operators. Operators with higher precedence are evaluated before those with lower precedence. This helps to avoid ambiguity and ensures that expressions are evaluated correctly.
- b) Associativity: Associativity comes into play when an expression contains multiple operators with the same precedence. It determines the order in which operators of the same precedence are evaluated. Operators can be left-associative (evaluated from left to right) or right-associative (evaluated from right to left).

Let's look at an example to understand these concepts better:

```
a = 10 + 5 * 2;
```

In this expression, the `*` operator has higher precedence than the `+` operator. Therefore, the multiplication ($5 * 2$) is performed first, resulting in 10. Then, the addition ($10 + 10$) is performed, resulting in the final value of 20.

Now, consider an example with operators of the same precedence and different associativity:

```
result = 8 / 4 / 2
```

Both the `/` operators have the same precedence. However, they are left-associative, so the expression is evaluated from left to right. The division is performed as follows: $(8 / 4) / 2$, resulting in a final value of 1.0.

In the C programming language, the memory required for integers (including long integers) and their range depend on the system architecture and data type used. The C standard doesn't specify the exact memory size for data types; it provides minimum ranges that data types should cover.

1. `int`: This is the basic integer type in C. It's typically 4 bytes (32 bits) on a 32-bit system and 4 or 8 bytes (32 or 64 bits) on a 64-bit system. The range of `int` is from `-2147483648` to `2147483647` on a 32-bit system and from `-2147483648` to `2147483647` or even wider on a 64-bit system.

2. `long int`: This is a longer integer type that can hold larger values than `int`. It's typically 4 bytes (32 bits) on a 32-bit system and 8 bytes (64 bits) on a 64-bit system. The range of `long int` is similar to `int` but can hold larger values.

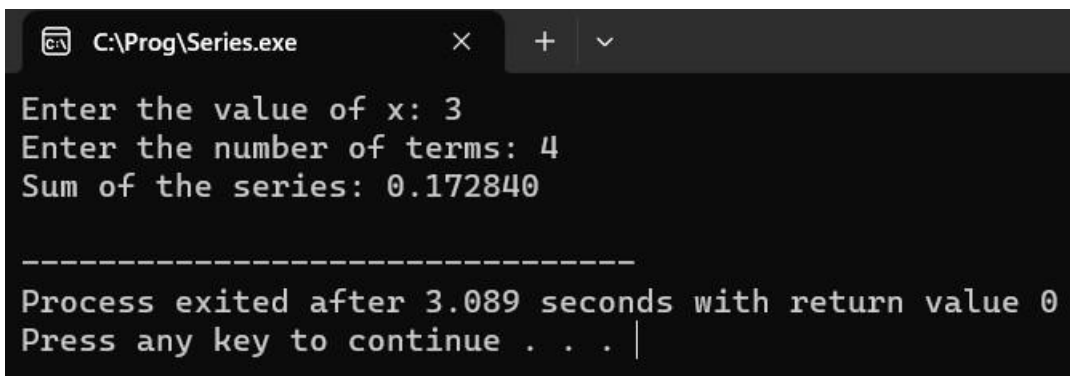
3. `long long int`: This is an even longer integer type introduced in the C99 standard. It's typically 8 bytes (64 bits) on both 32-bit and 64-bit systems. The range of `long long int` is from `-9223372036854775808` to `9223372036854775807`.

It's important to note that these are typical sizes and ranges, but the actual sizes and ranges can vary based on the compiler, system architecture, and platform. To find out the specific sizes and ranges on your system, you can use the `sizeof` operator to determine the memory size of a data type and refer to the limits defined in the `<limits.h>` header for the specific ranges.

2.b) Write a program in C to calculate the sum of given series up to the term given by the user. $Y = \frac{1}{2} - \frac{2}{x^2} + \frac{3}{x^3}$.

Ans:

```
#include <stdio.h>
#include <math.h>
int main () { int x, n;
float sum = 0.0;
    printf("Enter the value of x: "); scanf("%d",
&x);
    printf("Enter the number of terms: ");
scanf("%d", &n); for (int i=1; i<=n; i++) {
float term= i/ pow (x, i); if (i % 2 == 0) {
term *= -1;
    }
    sum += term;
}
    printf("Sum of the series: %f \n", sum);
return 0; }
```



```
C:\Prog\Series.exe
Enter the value of x: 3
Enter the number of terms: 4
Sum of the series: 0.172840

-----
Process exited after 3.089 seconds with return value 0
Press any key to continue . . . |
```

3.a) Distinguish between call by value and call by reference with example in C.

Ans: Call by Value and Call by Reference are two different parameter-passing mechanisms used in programming languages like C. They determine how arguments are passed to functions and how modifications to the arguments within the function affect the original values. Let's explore both concepts with examples in C.

Call by Value: In Call by Value, a copy of the actual arguments is passed to the function. The function works with these copies, and any changes made to the parameters within the function do not affect the original values outside the function.

Even though the value is modified inside the function, the original value remains unchanged in the main function. This is because a copy of it is passed to the function.

Call by Reference: In Call by Reference, a reference to the memory location of the actual arguments is passed to the function. This means that any changes made to the parameters within the function directly affect the original values outside the function.

Here, the value is modified inside the function, and this change directly affects the value in the main function because a reference to its memory location was passed.

In summary, Call by Value involves passing a copy of the argument, while Call by Reference involves passing a reference to the argument's memory location.

b) Write a program in C to find trace and norm of a matrix. Trace is defined as the sum of principal diagonal element; Norm is defined as square root of the sum of the square of all the elements in matrix.

Ans:

```
#include <stdio.h>
#include <math.h> #define
MAX_SIZE 10
double calculateNorm(double matrix[MAX_SIZE][MAX_SIZE], int size) {    double
norm = 0.0;
    for (int i = 0; i < size; i++) {        for (int j =
0; j < size; j++) {            norm += matrix[i][j]
* matrix[i][j];
        }
    }
    return sqrt(norm);
}
```

```

double calculateTrace(double matrix[MAX_SIZE][MAX_SIZE], int size) {
double trace = 0.0;   for (int i = 0; i < size; i++) {       trace += matrix[i][i];
    }
    return trace;
} int main ()
{   int size;
    printf("Enter the size of the square matrix (max %d): ", MAX_SIZE);   scanf("%d",
&size);

    if (size <= 0 || size > MAX_SIZE) {
printf("Invalid matrix size.\n");   return 1;
    }

    double matrix [MAX_SIZE] [MAX_SIZE];
printf("Enter the elements of the matrix:\n");   for (int i
= 0; i < size; i++) {       for (int j = 0; j < size; j++) {
scanf("%lf", &matrix[i][j]);
        }
    }

    double norm = calculateNorm(matrix, size);
double trace = calculateTrace(matrix, size);
printf("Matrix Norm: %lf\n", norm);   printf("Matrix
Trace: %lf\n", trace);   return 0;
}

```

```

C:\Prog\matrixxxx.exe
Enter the size of the square matrix (max 10): 2
Enter the elements of the matrix:
1
2
3
4
Matrix Norm: 5.477226
Matrix Trace: 5.000000

-----
Process exited after 10.51 seconds with return value 0
Press any key to continue . . .

```

4.a) Classify the variable according to the scope and extent (storage class) with C.

Ans: In C programming, variables can be classified based on their scope and storage class (extent). Scope refers to the region of the program where a variable is accessible, and storage class refers to the lifetime and visibility of a variable. There are four storage classes in C: `auto`, `static`, `extern`, and `register`. Let's go through each combination of scope and storage class along with explanations and examples:

1. Auto (Default Storage Class) - Local Variable:

Scope: Limited to the block or function where it is defined.

Storage Class: Automatic, allocated and deallocated automatically.

2. Static - Local Static Variable:

Scope: Limited to the block or function where it is defined.

Storage Class: Static, retains its value between function calls.

3. Register - Register Variable:

Scope: Limited to the block or function where it is defined.

Storage Class: Requests the compiler to store the variable in a register for faster access. The compiler may or may not honor the request.

4. Extern - External Variable:

Scope: Accessible across multiple source files.

Storage Class: Linked to a single storage location, visible across files.

Remember that local variables (auto, static, register) are defined within functions or blocks, while external variables (extern) are typically defined outside functions at the global scope. Additionally, the `static` storage class has different meanings for global variables. It limits the scope of the variable to the file it is defined in.

These concepts define how variables are stored, accessed, and maintained in C programs.

b) Write a program in C which calculates F, where $F = (a \cdot b^n) / c!$ and n is an integer. For designing above program, use two functions, one calculates factorial and other calculates power of a number.

Ans:

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

```

int calculateFactorial(int num) {
    if (num == 0 || num == 1) {
        return 1;
    } else {
        return num * calculateFactorial(num - 1);
    }
}

int calculatePower(int base, int exponent) {
    if (exponent == 0) {
        return 1;
    } else {
        return base * calculatePower(base, exponent - 1);
    }
}

int main() {
    int a, b, c;

    printf("Enter the values of a, b, and c: ");
    scanf("%d %d %d", &a, &b, &c);
    int factorialC = calculateFactorial(c);
    int powerB = calculatePower(b, c);
    float F = (float)(a * powerB) / factorialC;
    printf("F = %.2f\n", F);

    return 0;
}

```

```

C:\Prog\fact power.exe
Enter the values of a, b, and c: 2
3
4
F = 6.75

-----
Process exited after 3.931 seconds with return value 0
Press any key to continue . . .

```


5.a) Why pointer operator is called indirection operator? Give example.

Ans: The pointer operator, often denoted as `*`, is commonly referred to as the "indirection operator" because it is used to access the value stored at the memory location pointed to by a pointer. It "indirectly" refers to the value stored in memory through the pointer. In other words, it helps us navigate from the memory address (pointer) to the actual data stored at that address.

Consider the following example to understand the indirection operator:

```
#include <stdio.h>

int main() { int num
= 42; int *ptr;
ptr = &num;
printf ("Value of num: %d\n", num); printf
("Address of num: %p\n", &num); printf
("Value of ptr: %p\n", ptr);
printf ("Value pointed to by ptr: %d\n", *ptr); // Output: Value pointed to by ptr: 42 return 0;
}
```

In this example, ptr is a pointer that holds the address of the num variable. When we want to access the value stored in num using the pointer, we use the indirection operator *ptr, which means "the value pointed to by ptr". So, *ptr gives us the value `42`, which is the value stored in the memory location pointed to by ptr.

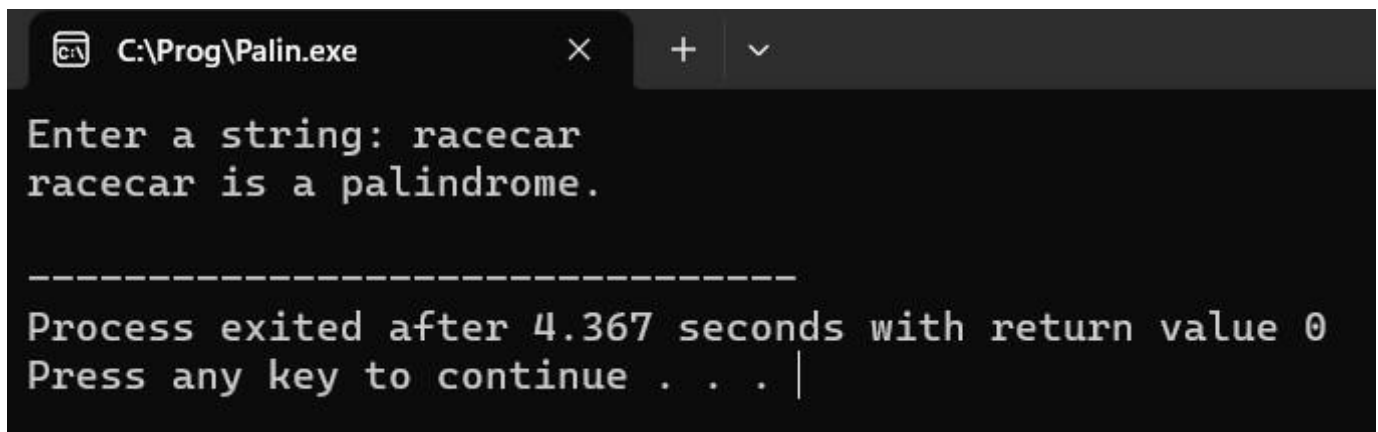
In summary, the indirection operator * allows us to indirectly access the value stored at a memory location through a pointer, making it a suitable name for the operator.

5.b) Write a C program to check whether the given string is palindrome or not. Palindrome should be checked by user defined function.

Ans:

```
#include <stdio.h>
#include <string.h> #include
<stdbool.h> bool
isPalindrome(char str[]) { int
length = strlen(str);
int i, j;
for (i = 0, j = length - 1; i < j; i++, j--) {
if (str[i] != str[j]) { return false;
}
```

```
    }  
    return true;  
} int main() {    char  
input[100];    printf("Enter a  
string: ");    scanf("%s",  
input);    if  
(isPalindrome(input)) {  
printf("%s is a palindrome.\n",  
input);  
    } else {  
    printf("%s is not a palindrome.\n", input);  
    }  
return 0; }
```



```
C:\Prog\Palin.exe  x  +  v  
Enter a string: racecar  
racecar is a palindrome.  
-----  
Process exited after 4.367 seconds with return value 0  
Press any key to continue . . . |
```

6.a) What are the differences between malloc () and calloc (). When we use realloc ().

Ans: malloc(), calloc(), and realloc() are all memory allocation functions in C, but they serve slightly different purposes. Here's a summary of their differences and when to use each one:

1. malloc()- Memory Allocation:

- malloc() stands for "memory allocation."
- It allocates a specified number of bytes of memory and returns a pointer to the first byte of the allocated memory.
- The content of the allocated memory is not initialized; it may contain garbage values.
- It takes a single argument, which is the size in bytes to allocate.

2. calloc() - Contiguous Allocation:

- calloc() stands for "contiguous allocation."
- It allocates memory for an array of elements, each with a specified size.
- The memory is initialized to zero, ensuring that all bits are set to 0.
- It takes two arguments: the number of elements and the size of each element in bytes.

3. realloc() - Reallocate Memory:

- realloc() stands for "reallocate."
- It is used to resize a previously allocated memory block, either increasing or decreasing its size.
- If the block is increased in size, the new space is uninitialized (similar to malloc()).
- If the block is decreased in size, data in the truncated memory is lost.
- It takes two arguments: a pointer to the previously allocated memory and the new size in bytes.

Usage Recommendations:

- Use malloc() when you need a specific amount of memory and you're going to initialize its content yourself.
- Use calloc() when you need a block of memory for an array and want the memory to be initialized with zeroes.
- Use realloc() when you need to change the size of an already allocated memory block, either to expand or shrink it.

Remember to free dynamically allocated memory using `free()` to avoid memory leaks.

Keep in mind that these functions might return a `NULL` pointer if the memory allocation fails, so always check the returned value before using the allocated memory.

6.b) Write a C program to read n integer numbers in an array dynamically and create two functions to sort in ascending and descending order and display the result in the main function.

Ans:

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

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void ascendingSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}

void descendingSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j
= 0; j < n - i - 1; j++) {
            if (arr[j] <
arr[j + 1]) {
                swap(&arr[j], &arr[j
+ 1]);
            }
        }
    }
}

int main() {
    int n;

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

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
```

```

printf("Memory allocation failed.\n");    return
1;
}
printf("Enter %d integer numbers:\n", n);    for (int
i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}
ascendingSort(arr, n);
printf("Array sorted in ascending order:\n");    for
(int i = 0; i < n; i++) {    printf("%d ", arr[i]);
}
printf("\n");    descendingSort(arr,
n);
printf("Array sorted in descending order:\n");    for
(int i = 0; i < n; i++) {    printf("%d ", arr[i]);
}
printf("\n");
free(arr);    return 0;
}

```

```

C:\Prog\sortinn.exe  ×  +  ▾
Enter the number of elements: 5
Enter 5 integer numbers:
1
2
3
4
5
Array sorted in ascending order:
1 2 3 4 5
Array sorted in descending order:
5 4 3 2 1

-----
Process exited after 7.818 seconds with return value 0
Press any key to continue . . . |

```

7. a) Discuss how array of strings can be stored in array of pointer variable.

Ans: Storing an array of strings in C involves using an array of pointers to char (i.e., an array of strings). Each element of the array points to the first character of a string, and the strings themselves are stored in separate memory locations.

Here's how an array of strings can be stored using an array of pointer variables:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> int
main() { char
*names[] = {
    "Alice",
    "Bob",
    "Charlie",
    "David"
};
    printf("First name: %s\n", names[0]);    printf("Second
name: %s\n", names[1]);    printf("Third name: %s\n",
names[2]);    printf("Fourth name: %s\n", names[3]);
strcpy(names[1], "Robert");
    printf("Modified second name: %s\n", names[1]);    return 0;
}
```

In this example, `names` is an array of pointers to char. Each element of the array (`names[0]`, `names[1]`, etc.) is a pointer pointing to the first character of a string literal. You can also allocate memory dynamically using `malloc` to store strings and then assign the pointers to those dynamically allocated memory locations.

Keep in mind the following important points:

1. String literals are stored in read-only memory, so modifying them directly can lead to undefined behavior. In the example above, we shouldn't modify the string literals directly, but rather use dynamic memory allocation if you need to modify strings.

2. When working with dynamic memory allocation for strings, don't forget to free the allocated memory using `free` to avoid memory leaks
3. The length of each string can be different, allowing you to store strings of varying lengths in the same array.

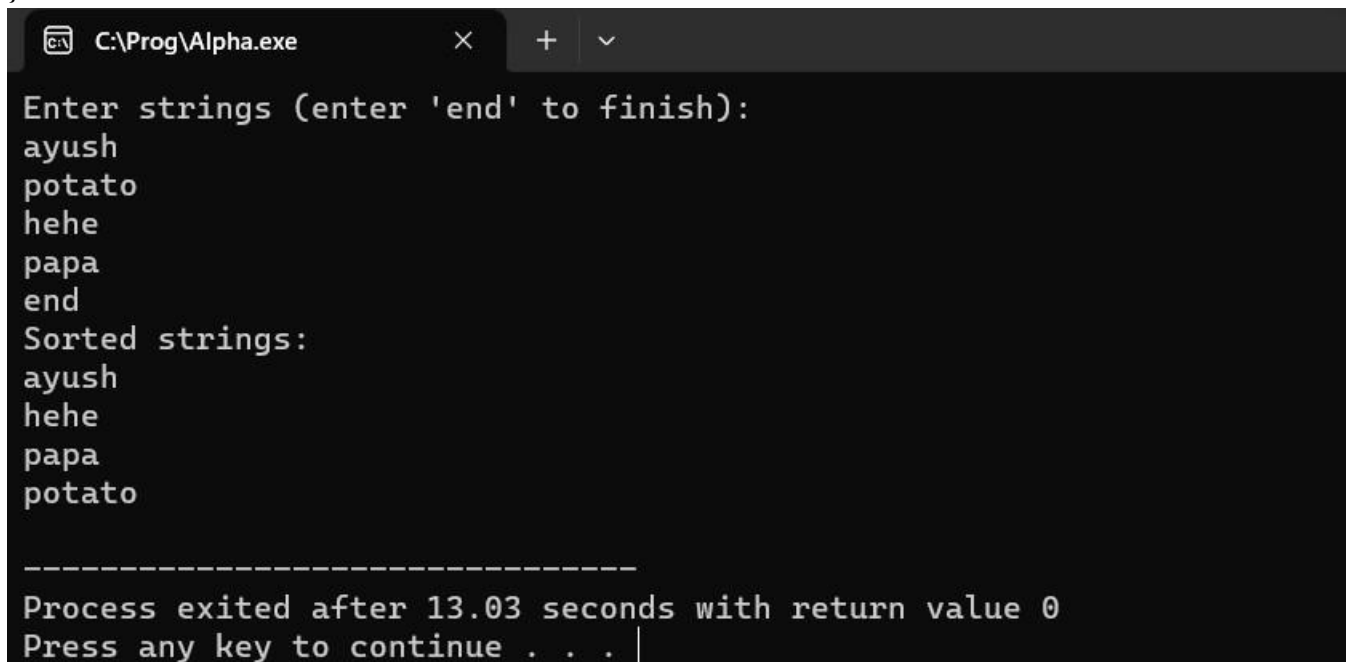
By using an array of pointer variables, you can create and manipulate an array of strings in C.

7.b) Write a C program to enter the strings until the user enters "end" and display the list of string in alphabetical order using two-dimensional array of character using function.

Ans)

```
#include <stdio.h>
#include <string.h>
void sortStrings(char strings[][100], int n) {
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (strcmp(strings[i], strings[j]) > 0)
            {
                char temp[100];
                strcpy(temp, strings[i]);
                strcpy(strings[i], strings[j]);
                strcpy(strings[j], temp);
            }
        }
    }
} int main()
{
    char strings[100][100];
    int n = 0;
    printf("Enter strings (enter 'end' to finish):\n");
    while
(1) {
        scanf("%s", strings[n]);
        if
(strcmp(strings[n], "end") == 0) {
            break;
        }
    }
}
```

```
        n++;
    }
    sortStrings(strings, n);
    printf("Sorted strings:\n");    for (int
i = 0; i < n; i++) {
    printf("%s\n", strings[i]);
    }
    return 0;
}
```



```
C:\Prog\Alpha.exe  x  +  v
Enter strings (enter 'end' to finish):
ayush
potato
hehe
papa
end
Sorted strings:
ayush
hehe
papa
potato

-----
Process exited after 13.03 seconds with return value 0
Press any key to continue . . . |
```


8. a) Explain nested structure in C, with example.

Ans: In C programming, a nested structure is a structure that is defined within another structure. This means that you can have a structure member of another structure type. This concept allows you to create more complex data structures by grouping related data together. Nested structures are particularly useful when you need to represent hierarchical or composite data.

```
#include <stdio.h> struct Date { int day; int month; int year;
```

```
};
```

```
struct Student {
```

```
int rollNumber;
```

```
char name[50];
```

```
struct Date birthdate;
```

```
};
```

```
int main() {
```

```
struct Student student1;
```

```
printf("Enter student's roll number: ");
```

```
scanf("%d", &student1.rollNumber);
```

```
printf("Enter student's name: ");
```

```
scanf("%s", student1.name);
```

```
printf("Enter student's birth date (day month year): ");
```

```
scanf("%d %d %d", &student1.birthdate.day, &student1.birthdate.month, &student1.birthdate.year);
```

```
printf("\nStudent Information:\n");
```

```
printf("Roll Number: %d\n", student1.rollNumber);
```

```
printf("Name: %s\n", student1.name);
```

```
printf("Birth Date: %d/%d/%d\n", student1.birthdate.day, student1.birthdate.month, student1.birthdate.year);
```

```
return 0;
```

```
}
```

8.b) Write a C program that illustrates the pointer pointing to the function.

Ans:

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

```
int add(int a, int b) {
```

```
return a + b;
```

```
}
```

```
int subtract(int a, int b)
```

```
{
```

```
return a - b;
```

```
}
```

```
int main()
```

```
{
```

```
int (*operation)(int, int);
```

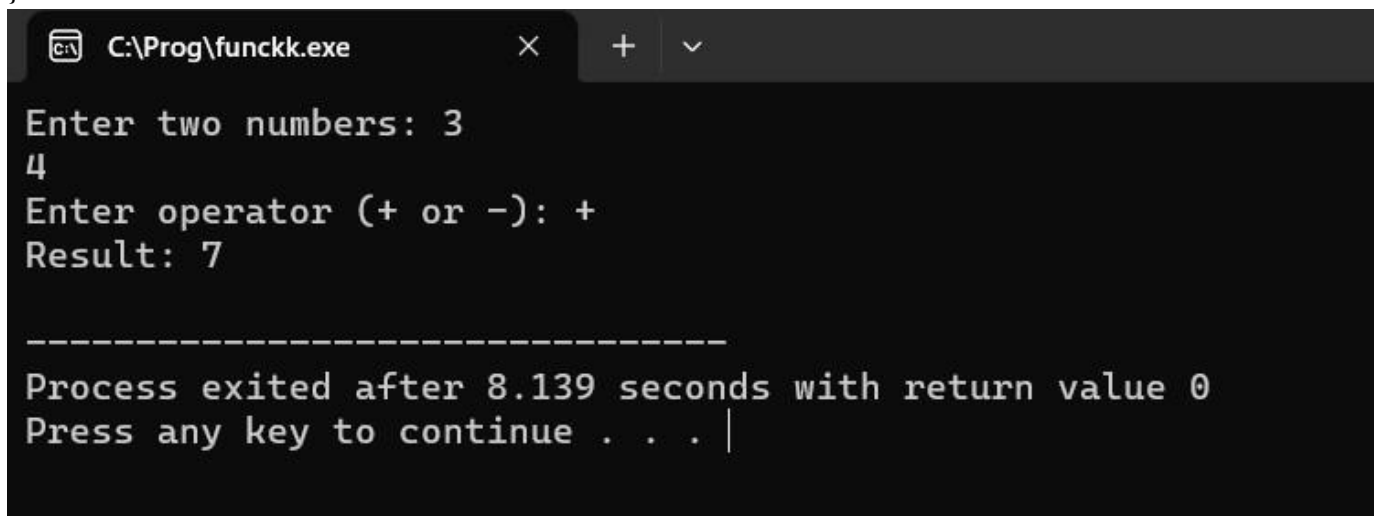
```
int num1, num2;
```

```
char operate;
```

```
printf("Enter two numbers: ");
```

```
scanf("%d %d", &num1, &num2);
```

```
printf("Enter operator (+ or -): ");
scanf(" %c", &operate);
if (operate == '+') {
    operation = add;    }
else if (operate == '-') {
    operation = subtract;
    }
else
{
    printf("Invalid operator\n");
return 1;
    }
    int result = operation(num1, num2);
printf("Result: %d\n", result);
return 0;
}
```



```
C:\Prog\funck.exe  x  +  v
Enter two numbers: 3
4
Enter operator (+ or -): +
Result: 7

-----
Process exited after 8.139 seconds with return value 0
Press any key to continue . . . |
```

9) How can we pass structure variable as a parameter in function? Illustrate with an example.

Answer:

C allows passing of structure as arguments to function.

There are 2 methods by which values of structure can be transferred from one function to another.

1. pass by value

2. pass by reference

1. Pass by value

It involves passing a copy of the entire structure to the called function. Any changes to the structure members within the function is not reflected in the original structure (in the calling function). Therefore, it becomes necessary for the function to return entire structure back to the calling function.

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

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

```
struct student
```

```
{
```

```
char name[50];
```

```
int roll;
```

```
float marks;
```

```
};
```

```
void display(struct student);
```

```
int main()
```

```
{
```

```
struct student s;
```

```
printf("Enter name of student:");
```

```
gets(s.name);
```

```
printf("Enter roll number:");
```

```
scanf("%d",&s.roll);
```

```
printf("Enter marks:");
```

```
scanf("%f",&s.marks);
```

```
display(s);
```

```
getch();
```

```

return 0;
}
void display(struct student st)
{
printf("Name=%s\nRoll=%d\nMarks=%f",st.name,st.roll,st.marks);
}

```

2. Passing by reference

It uses the concept of pointer to pass structure as an argument. The address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This method is more efficient than the first one. Structure pointer operator is used to access the members.

```

#include<stdio.h>
void swapNum(int*i, int*j){
int temp=*i;
*i=*j;
*j=temp;
}
int main(void)
{
int a=10;
int b=20;
swapNum(&a,&b);
printf("A is %d and B is %d\n",a,b);
return 0;
}

```

b) Create a structure Student(Roll, Name , Address, Age) to store 10 different records of students and another structure Mark_Sheet (SN, Subject_Name, Full_Marks, Pass_Marks, Marks_Obtained) within student to store the record of 5 different subjects. WAP to enter the record of students and display the records with the percentage each student have scored

```

#include <stdio.h>

```

```
#include<string.h>
struct Mark_Sheet{
int SN;
char Subject_Name[30];
int fullMarks;
int passMarks;
int obtainedMarks;

}ms;
struct Student{
int Roll;
char name[50];
char Address[100];
int age;
struct Mark_Sheet ms[5];

}s;
int main(){
    struct Student s[10];

for(int i=0;i<10;i++){
    printf("Enter details for students %d:\n",i+1);
    printf("Roll");
    scanf("%d",&s[i].Roll);
    printf("Name:");
    scanf("%s",s[i].name);
    printf("Address:");
    scanf("%s",s[i].Address);
    printf("Age:");
    scanf("%d",&s[i].age);
```

```

for(int j=0;j<5;j++){
    printf("Enter details for subjects %d\n",j+1);
    s[i].ms[j].SN=j+1;
    printf("Subject Name:");
    scanf("%s",s[i].ms[j].Subject_Name);
    printf("Full Marks:");
    scanf("%d",&s[i].ms[j].fullMarks);
    printf("Pass Marks:");
    scanf("%d",&s[i].ms[j].passMarks);
    printf("Marks Obtained:");
    scanf("%f",&s[i].ms[j].obtainedMarks);
}
}
for(int i=0;i<10;i++){
    printf("\n Student %d Details:\n",i+1);
    printf("Roll %d\n",s[i].Roll);
    printf("Name: %s\n",s[i].name);
    printf("Address: %s\n",s[i].Address);
    printf("Age is %d\n",s[i].age);
    int tm=0; //total marks
    float to=0; //total obtained
    for(int j=0;j<5;j++){
        printf("Subject %d:\n",j+1);
        printf("Subject Name: %s\n",s[i].ms[j].Subject_Name);
        printf("Full Marks:%d\n",s[i].ms[j].fullMarks);
        printf("Pass Marks: %d\n",s[i].ms[j].passMarks);
        printf("Marks Obtained : %f\n",s[i].ms[j].obtainedMarks);
    }
    float percentage=(to/tm)*100;
    printf("Total marks obtained is %f\n",to);
    printf("Total marks %d\n",tm);
}

```

```
printf("Percentage %.2f\n",percentage);  
}  
return 0;  
}
```

10) In which condition unary operator is equivalent to assignment operator. Explain with examples.

A unary operator can be equivalent to an assignment operator under certain conditions when used in a specific context. This happens when a unary operator modifies a variable's value and assigns the modified value back to the same variable. The compound assignment operators, such as `+=`, `-=` etc., are good examples of this behavior.

For example;

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

```
int main() {
```

```
int num = 5;
```

```
num += 3; // Equivalent to num = num + 3;
```

```
printf("num = %d\n", num); // Output: num = 8
```

```
return 0;
```

```
}
```

b) WAP in C to create a file named "sub.txt" that stores subject name, subject code and full marks where subject name and subject code are string and full marks in float. Also display all the records stored in a file.WAP to keep the records of 10 subjects and display all record from file.

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

```
struct Subject {
```

```
char name[50];
```

```
char code[20];
```

```
float fullMarks;
```



```
}s;
```

```
int main() {  
    FILE *file;  
    struct Subject s[10];  
  
    // Input subject records  
    for (int i = 0; i < 10; i++) {  
        printf("Enter details for subject %d:\n", i + 1);  
        printf("Subject Name: ");  
        scanf("%s", s[i].name);  
        printf("Subject Code: ");  
        scanf("%s", s[i].code);  
        printf("Full Marks: ");  
        scanf("%f", &s[i].fullMarks);  
    }  
  
    // Create and write subject records to the file  
    file = fopen("sub.txt", "w");  
    if (file == NULL) {  
        printf("Error opening file.\n");  
        return 1;  
    }  
  
    for (int i = 0; i < 10; i++) {  
        fprintf(file, "Subject Name: %s\n", s[i].name);  
        fprintf(file, "Subject Code: %s\n", s[i].code);  
        fprintf(file, "Full Marks: %.2f\n", s[i].fullMarks);  
    }  
  
    fclose(file);  
}
```

```
// Display subject records from the file
file = fopen("sub.txt", "r");
if (file == NULL) {
    printf("Error opening file.\n");
    return 1;
}

char line[100];
printf("\nSubject Records:\n");
while (fgets(line, sizeof(line), file)) {
    printf("%s", line);
}

fclose(file);

return 0;
}
```

2067 Ashwin Solutions

1) Define interpreter and compiler. Briefly explain the steps followed for developing computer software.

A compiler is a program that translates a program written in High level language to executable machine language. The process of transferring High level source program in to object code is a lengthy and complex process as compared to assembling. The basic purpose of interpreter is same as that of compiler. In compiler, the program is translated completely and directly executable version is generated. Whereas interpreter translates each instruction, executes it and then the next instruction is translated and this goes on until end of the program.

The steps for developing computer software are:

1. Requirement Gathering:

- Identify and gather the specific requirements for the software from stakeholders, end-users, and any other relevant parties.
- Document functional and non-functional requirements, user stories, and use cases to define what the software needs to accomplish.

2. Analysis:

- Analyze the gathered requirements to understand the scope, complexity, and potential challenges of the project.
- Define the system architecture, data models, and overall design of the software.

3. Design:

- Create a detailed design of the software, including high-level and low-level design components.
- Decide on the software's structure, user interfaces, databases, algorithms, and other technical aspects.

4. Implementation / Coding:

- Write the actual code based on the design specifications.
- Follow coding standards, best practices, and guidelines to ensure maintainability and readability of the code.
- Regularly review and test the code for bugs and errors during the coding process.

5. Testing:

- Develop a comprehensive testing plan to verify that the software meets the specified requirements.
- Conduct unit testing to check individual components or modules.
- Perform integration testing to ensure different parts of the software work together as expected.
- Execute system testing to assess the software's overall functionality and performance.
- Carry out user acceptance testing (UAT) with actual users to validate that the software meets their needs.

6. Deployment:

- Prepare the software for deployment in the target environment, which could be on-premises servers or cloud platforms.
- Install and configure the software, ensuring all necessary dependencies are in place.
- Address any deployment-related issues that may arise.

7. Maintenance and Support:

- Once the software is deployed, provide ongoing maintenance, bug fixes, and updates as needed.
- Gather user feedback to identify potential improvements and enhancements.
- Continuously monitor the software's performance and address any issues that may arise.

8. Documentation:

- Create comprehensive documentation that explains the software's functionality, architecture, design decisions, and usage instructions.
- This documentation helps developers, testers, and users understand the software and its features.

9. Training and User Support:

- Provide training to end-users, administrators, and support personnel on how to use and manage the software effectively.

10. Iterative Process:

- Many software development methodologies, such as Agile, emphasize iterative and incremental development.
- This involves repeating some or all of the above steps in cycles to continually improve the software based on feedback and changing requirements.

2) What is data type? Explain the operators available in C along with their precedence and associativity.

The data type in C defines the amount of storage allocated to variables, the values that they can accept, and the operation that can be performed on those variables. C is rich in data types. The variety of data types allow the programmer to select appropriate data type to satisfy the need of application as well as the needs of different machine. There are following type of data types supported by c programming

- Primary Data Type
- Derived Data Type
- User Defined Data Type

The data type and the value of an expression depends on the data types of the operands and the order of evaluation of operators which is determined by the precedence and associativity of operators.

1. Postfix Operators:

- () Function call
- [] Array subscript
- -> Structure/Union member access (by pointer)
- . Structure/Union member access (by value)
- (type) Type casting

Associativity: Left to right

2. Unary Operators:

- ++ Increment
- -- Decrement
- + Unary plus
- - Unary minus
- ! Logical NOT
- ~ Bitwise NOT
- & Address-of
- * Dereference (value at address)

- **sizeof** Size in bytes of a type or expression
- **(type)** Type casting

Associativity: Right to left

3. Multiplicative Operators:

- * Multiplication
- / Division
- % Modulus (remainder)

Associativity: Left to right

4. Additive Operators:

- + Addition
- - Subtraction

Associativity: Left to right

5. Shift Operators:

- << Left shift
- >> Right shift

Associativity: Left to right

6. Relational Operators:

- < Less than
- > Greater than
- <= Less than or equal to
- >= Greater than or equal to

Associativity: Left to right

7. Equality Operators:

- == Equal to
- != Not equal to

Associativity: Left to right

8. Bitwise Operators:

- & Bitwise AND
- ^ Bitwise XOR (exclusive OR)
- | Bitwise OR

Associativity: Left to right

9. Logical Operators:

- && Logical AND
- || Logical OR

Associativity: Left to right

10. Conditional Operator:

- ? : Ternary conditional (conditional expression)

Associativity: Right to left

11. Assignment Operators:

- = Assignment
- += Addition assignment
- -= Subtraction assignment
- *= Multiplication assignment
- /= Division assignment
- %= Modulus assignment
- <<= Left shift assignment
- >>= Right shift assignment
- &= Bitwise AND assignment

- ^= Bitwise XOR assignment
- |= Bitwise OR assignment

Associativity: Right to left

12. Comma Operator:

- , Comma separator (used in expressions)

Associativity: Left to right

- 3) You are given a task to develop a system to read at least 50 integer numbers and continue until the user enters NO. Your system must have capacity to calculate sum and average of those numbers which are exactly divisible by 9 but not by 6 and lies between 1 to 100 and display a suitable message if no such number is read. Write algorithm.

```
#include <stdio.h>

int main() {
    int num, count = 0, sum = 0;
    double average;

    printf("Enter integer numbers between 1 and 100 (Enter NO to stop):\n");

    while (1) {
        char input[3]; // Assuming "NO" as input, plus null character
        printf("Enter a number or NO: ");
        scanf("%s", input);

        if (strcmp(input, "NO") == 0) {
            break;
        }

        num = atoi(input);

        if (num >= 1 && num <= 100) {
            if (num % 9 == 0 && num % 6 != 0) {
                sum += num;
                count++;
            }
        } else {
            printf("Number must be between 1 and 100.\n");
        }
    }

    if (count > 0) {
        average = sum / count;
        printf("Sum of numbers divisible by 9 but not by 6: %d\n", sum);
    }
}
```

```
    printf("Average of these numbers: %.2f\n", average);
} else {
    printf("No valid numbers were entered.\n");
}

return 0;
}
```

Algorithm

1. Initialize variables: num, sum = 0, count = 0
2. Display a message to instruct the user to enter integer numbers between 1 and 100 (Enter NO to stop)
3. Repeat the following steps indefinitely:
 - a. Display a prompt to enter a number or "NO"
 - b. Read input as a string and store it in input
 - c. If input is equal to "NO", exit the loop
 - d. Convert input to an integer using atoi function, store it in num
 - e. If num is within the range of 1 to 100, proceed to the next step, else display an error message
 - f. If num is divisible by 9 and not divisible by 6, update sum by adding num and increment count
4. If count is greater than 0, calculate the average as $\text{sum} / \text{count}$
5. Display the sum and average if count > 0, else display a message that no valid numbers were entered
6. End

4) **How recursion is different from iteration? Write code in C to calculate the sum of following series up to n terms specified by the user where n is passed to the function that calculates the sum. Your program should have more than two functions:**

$(2*3/5)+(4*5/7)+(6*7/9)+\dots\dots\dots$

Recursion: Recursion is a programming technique where a function calls itself in order to solve a problem. In a recursive function, the problem is divided into smaller sub problems, and the function is applied to those sub problems. Each recursive call reduces the original problem's size until it reaches a base case, which is a simple scenario where the function doesn't call itself again and returns a result.

Iteration: Iteration is the process of repeatedly executing a set of statements in a loop until a certain condition is met. It involves using loop constructs (like for, while, or do-while) to control the repetition of code. Iteration is suitable for tasks where a specific number of iterations is known or where the problem does not naturally lend itself to recursive decomposition.

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

```
float calculateSeriesTerm(int term) {  
    return ((2.0 * term) * (2.0 * term + 1)) / (2.0 * term + 3);  
}
```

```
float calculateSeriesSum(int n) {  
    float sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += calculateSeriesTerm(i);  
    }  
    return sum;  
}
```

```
void displaySeriesAndSum(int n, float sum) {  
    printf("Series up to %d terms:\n", n);  
    for (int i = 1; i <= n; i++) {  
        printf("(%d*%d/%d)", 2 * i, 2 * i + 1, 2 * i + 3);  
        if (i < n) {  
            printf(" + ");  
        }  
    }  
}
```



```

    printf("\nSum of the series up to %d terms: %.4f\n", n, sum);
}

int main() {
    int n;

    printf("Enter the number of terms: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Number of terms should be positive.\n");
    } else {
        float result = calculateSeriesSum(n);
        displaySeriesAndSum(n, result);
    }

    return 0;
}

```

5) Describe the limitations of using `getchar` and `scanf` functions for reading strings. Write a complete program to insert a string into another string in the location specified by the user. Read the string in main function along with inserting position and returns the resulting strings.

1. **Spaces and Newlines:** Both `getchar()` and `scanf()` treat spaces and newline characters as delimiters, which can make it challenging to read strings containing spaces or multiple lines.
2. **Buffer Overflow:** The buffer size allocated by default for reading strings using `getchar()` or `%s` in `scanf()` can lead to buffer overflow issues if the input string is longer than the buffer size.
3. **No Control Over Buffer Size:** With `getchar()`, you have no direct control over the buffer size, and with `scanf()`, you can specify a buffer size, but it's limited and might not be sufficient for all cases.
4. **String Termination:** `getchar()` does not automatically append the null-terminating character, requiring additional code to terminate the string. `scanf()` appends a null-terminator, but it may not work as expected when reading multiple strings.
5. **Limited Formatting Control:** `scanf()` can have issues with newline characters from previous inputs remaining in the buffer, leading to unexpected behavior.

```

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

int main()
{
    char a[10];
    char b[10];

```

```

char c[10];
int p=0,r=0,i=0;
int t=0;
int x,g,s,n,o;

puts("Enter First String:");
gets(a);
puts("Enter Second String:");
gets(b);
printf("Enter the position where the item has to be inserted: ");
scanf("%d",&p);
r = strlen(a);
n = strlen(b);
i=0;

// Copying the input string into another array
while(i <= r)
{
    c[i]=a[i];
    i++;
}
s = n+r;
o = p+n;

// Adding the sub-string
for(i=p;i<s;i++)
{
    x = c[i];
    if(t<n)
    {
        a[i] = b[t];
        t=t+1;
    }
    a[o]=x;
    o=o+1;
}

printf("%s", a);
return 0;
}

```

- 6) **Explain the relation of array and pointer. What is meant by call by value and call by reference? Write a complete program that adds corresponding elements of two matrices if the elements are positive, otherwise multiply the corresponding elements using the concept of passing array to function and pointer.**

In C programming, pointers and array shares a very close relationship. Array is a data structure that hold finite sequential collection of similar type data. We use array to store a collection of similar type data together. To access and array element we use index. These index starts from 0 and goes up to N-1 (where N is size of the array).

Call By Value

In this particular parameter passing method, the values of the actual parameters copy into the function's formal parameters. It stores both types of parameters in different memory locations. Thus, if one makes any changes inside the function- it does not show on the caller's actual parameters.

This method passes a copy of an actual argument to the formal argument of any called function. In the case of a Call by Value method, any changes or alteration made to the formal arguments in a called function does not affect the overall values of an actual argument. Thus, all the actual arguments stay safe, and no accidental modification occurs to them.

Call by Reference

In this case, both the formal and actual parameters refer to a similar location. It means that if one makes any changes inside the function, it gets reflected in the caller's actual parameters.

This method passes the address or location of the actual arguments to the formal arguments of any called function. It means that by accessing the actual argument's addresses, one can easily alter them from within the called function. Thus, in Call by Reference, it is possible to make alterations to the actual arguments. Thus, the code needs to handle the arguments very carefully. Or else, there might be unexpected results and accidental errors.

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

```
// Function to perform the operation on matrices
```

```
void operateMatrices(int rows, int cols, int mat1[][cols], int mat2[][cols], int result[][cols], int (*operation)(int, int)) {  
    for (int i = 0; i < rows; i++) {  
        for (int j = 0; j < cols; j++) {  
            result[i][j] = operation(mat1[i][j], mat2[i][j]);  
        }  
    }  
}
```

```

}

// Addition function
int add(int a, int b) {
    return a + b;
}

// Multiplication function
int multiply(int a, int b) {
    return a * b;
}

// Display matrix function
void displayMatrix(int rows, int cols, int mat[][cols]) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int rows, cols;

    printf("Enter the number of rows and columns: ");
    scanf("%d %d", &rows, &cols);

    int mat1[rows][cols], mat2[rows][cols], result[rows][cols];

    printf("Enter elements of matrix 1:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &mat1[i][j]);
        }
    }

    printf("Enter elements of matrix 2:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &mat2[i][j]);
        }
    }
}

```

```
operateMatrices(rows, cols, mat1, mat2, result, add); // Perform addition operation
printf("Result after addition:\n");
displayMatrix(rows, cols, result);
```

```
operateMatrices(rows, cols, mat1, mat2, result, multiply); // Perform multiplication operation
printf("Result after multiplication:\n");
displayMatrix(rows, cols, result);
```

```
return 0;
```

```
}
```

7) Define structures in C programming. Explain nested structures with suitable example. Write a program using structure and passing the structures to the function that returns the results to convert the date in format(YY/MM/DD) in BS to AD.

A structure is a composite data type that groups together variables under a single name. Each variable within the structure is called a member or field, and these members can have different data types. Structures allow you to create custom data types to represent entities with multiple attributes.

Nested Structure

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

```
// Outer structure
```

```
struct Date {
    int day;
    int month;
    int year;
};
```

```
// Inner structure
```

```
struct Student {
    int rollNumber;
    char name[50];
    struct Date birthdate; // Nested structure
};
```

```
int main() {
```

```
    struct Student student1;
```

```
    student1.rollNumber = 101;
```

```
    strcpy(student1.name, "John Doe");
```

```
    student1.birthdate.day = 5;
```

```
    student1.birthdate.month = 12;
```

```
    student1.birthdate.year = 2000;
```

```

printf("Student Information:\n");
printf("Roll Number: %d\n", student1.rollNumber);
printf("Name: %s\n", student1.name);
printf("Birthdate: %d-%d-%d\n",
       student1.birthdate.day,
       student1.birthdate.month,
       student1.birthdate.year);

return 0;
}

```

Code

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

```

struct DateBS {
    int year;
    int month;
    int day;
};

```

```

struct DateAD {
    int year;
    int month;
    int day;
};

```

```

int isLeapYearBS(int year) {
    return ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0);
}

```

```

void convertBSToAD(struct DateBS bsDate, struct DateAD *adDate) {
    int daysInMonthBS[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    if (isLeapYearBS(bsDate.year)) {
        daysInMonthBS[2] = 29;
    }

    int totalDays = 0;

    for (int i = 1; i < bsDate.year; i++) {
        totalDays += isLeapYearBS(i) ? 366 : 365;
    }
}

```

```

for (int i = 1; i < bsDate.month; i++) {
    totalDays += daysInMonthBS[i];
}

totalDays += bsDate.day - 1;

adDate->year = 1943; // AD equivalent of 2000 BS
adDate->month = 4;
adDate->day = 14;

while (totalDays > 0) {
    if (isLeapYearBS(adDate->year)) {
        totalDays -= 366;
    } else {
        totalDays -= 365;
    }
    adDate->year++;
}

int daysInMonthAD[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

if ((adDate->year % 4 == 0 && adDate->year % 100 != 0) || (adDate->year % 400 == 0)) {
    daysInMonthAD[2] = 29;
}

int month = 1;
while (totalDays >= daysInMonthAD[month]) {
    totalDays -= daysInMonthAD[month];
    month++;
}

adDate->month = month;
adDate->day = totalDays + 1;
}

int main() {
    struct DateBS bsDate;
    struct DateAD adDate;

    printf("Enter the date in BS (Year Month Day): ");
    scanf("%d %d %d", &bsDate.year, &bsDate.month, &bsDate.day);

    convertBSToAD(bsDate, &adDate);
}

```

```
printf("Converted AD Date: %d/%02d/%02d\n", adDate.year, adDate.month, adDate.day);

return 0;
}
```

8) Explain the I/O operations on file with suitable examples and also state the typical error situations during I/O operations in file. How the contents in the file can be randomly accessed?

Input/Output (I/O) operations on files in C involve reading from and writing to files. Files are a way to store data persistently on disk, and C provides various functions for performing these operations.

I/O Operations on Files:

1. **Opening a File:** The `fopen()` function is used to open a file. It takes two arguments: the file name and the mode (e.g., "r" for read, "w" for write, "a" for append).

Example:

```
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    printf("Failed to open the file.\n");
    return 1;
}
```

Reading from a File: The `fscanf()` function is used to read data from a file. It is similar to `scanf()` for standard input.

Example:

```
int num;
fscanf(file, "%d", &num);
```

Writing to a File: The `fprintf()` function is used to write data to a file. It is similar to `printf()` for standard output.

Example

```
fprintf(file, "Hello, world!\n");
```

Closing a File: The `fclose()` function is used to close an opened file.

Example:

```
fclose(file);
```

Typical Error Situations During I/O Operations:

1. **File Not Found:** If the file does not exist while trying to open it for reading, `fopen()` returns `NULL`.
2. **Permission Issues:** If the file does not have proper permissions for reading or writing, the I/O operations will fail.
3. **Disk Full:** When writing to a file, if the disk is full or the user's quota is exceeded, writing will fail.
4. **Reading/Writing Errors:** Reading or writing errors can occur due to various reasons such as hardware issues, interruption, or corrupted files.

Random Access to File Contents:

C provides the `fseek()` and `ftell()` functions for random access to file contents.

- `fseek(file, offset, origin)` sets the file position indicator to a new position based on the origin (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`) and offset (number of bytes from origin).
- `ftell(file)` returns the current file position indicator.

Example of random access:

```
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    printf("Failed to open the file.\n");
    return 1;
}

fseek(file, 10, SEEK_SET); // Move 10 bytes from the beginning

char c = fgetc(file); // Read a character from the new position

fclose(file);
```

9. What are FORTRAN's constants, variables and library functions? Explain. Write a program in FORTRAN to take position and a number and insert this number on this position inside an array containing n elements. [4+8]

Answer

A constant has a value which is fixed when the program is written. It is called constant as its value doesn't change during program execution. There are 5 types of constants in FORTRAN:

- i. Integer constants: -10, 420, -69, etc
- ii. Real constants: A real constant contains a decimal point or exponent. It is similar to float data type in C.
- iii. Complex constants: A complex constant consists of two components; the first component is the real component whereas the second is the imaginary component. eg., (2.14, -3.67)
- iv. Logical constants: TRUE and FALSE are the only logical constants and are expressed as .TRUE and .FALSE.
- v. Character constants: A character constant consists of a string of characters enclosed in a pair of apostrophes which act as quotation marks, eg. 'Ram', 'A'.

A variable is an identifier with a name, data type, and value. Its type depends upon either data type provided at the time of its declaration (i.e. explicit definition) or first character in its name (i.e. in the case of implicit definition). A variable cannot be used or referred to unless it has been defined through an assignment statement, input statement, or through association with a variable or array element that has been defined.

The library functions are built in functions which can be directly called in program to make program development easy. Some of the library functions available in FORTRAN are:

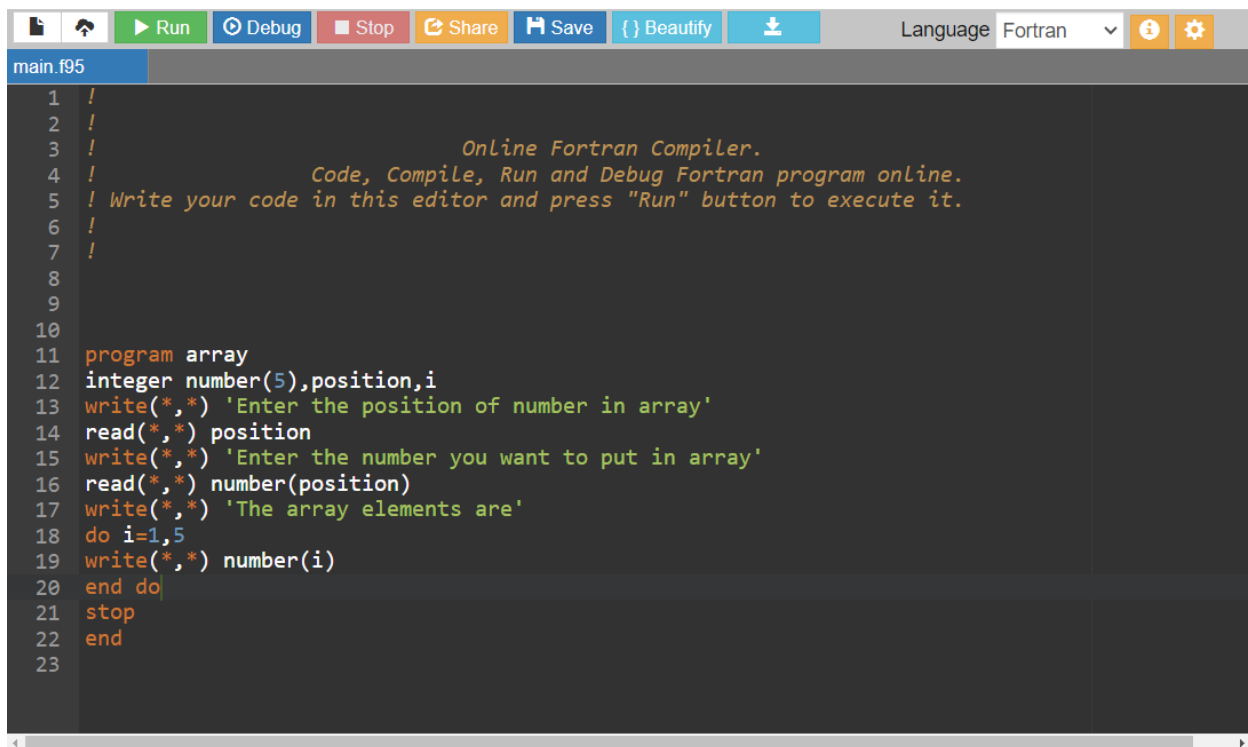
Abs (): To calculate absolute value

Min (): To calculate minimum value among multiple numbers

Max (): To calculate maximum value among multiple numbers

Read (): For reading data from input device

Write (): For writing data to output device



The screenshot shows an online Fortran compiler interface. At the top, there is a toolbar with buttons for Run, Debug, Stop, Share, Save, Beautify, and a download icon. The language is set to Fortran. The main editor area contains the following Fortran code:

```
1 !
2 !
3 !           Online Fortran Compiler.
4 !           Code, Compile, Run and Debug Fortran program online.
5 ! Write your code in this editor and press "Run" button to execute it.
6 !
7 !
8
9
10
11 program array
12 integer number(5),position,i
13 write(*,*) 'Enter the position of number in array'
14 read(*,*) position
15 write(*,*) 'Enter the number you want to put in array'
16 read(*,*) number(position)
17 write(*,*) 'The array elements are'
18 do i=1,5
19 write(*,*) number(i)
20 end do
21 stop
22 end
23
```

```
input
Enter the position of number in array
4
Enter the number you want to put in array
6
The array elements are
    0
    0
    0
    6
    0

...Program finished with exit code 0
Press ENTER to exit console.
```

Compiler used: https://www.onlinegdb.com/online_fortran_compiler

2067 Ashad Solutions

1. What is high level language? What are the different types of high level languages? How computer programming language C is different from FORTRAN? [1+3+4]

Answer

A high-level language is any programming language that enables development of a program in a much more user-friendly programming context and is generally independent of the computer's hardware architecture. A high level language can be further categorized into object oriented and procedural programming language.

Procedure Oriented Programming Language

Procedural programming is based upon the idea of series of procedure calls. A programmer writing program in this language can exactly specify a sequence of steps in order to perform a particular task. Languages which express step-by-step algorithms written to solve a problem are known as procedural languages. A procedure may be a program in itself that may be called within a main program, a subroutine or another program. A programmer knows exactly what is to be accomplished at the end of the program and uses a sequence of algorithmic steps in order to achieve it. C language is example of procedure oriented programming language.

Object Oriented Programming Language

In object oriented programming; the data (i.e. variables) and code are combined to form objects. This allows more effective code duplication which is not the case when programs are divided into subroutines (or functions). The required programming parts can be called again and again within the program. Its main distinction with procedural programming is that in procedural programming task is divided into subroutines, structures and functions. Whereas in object oriented programming, data as well as functions and subroutines are encapsulated to form objects. C++ language is example of object oriented programming language.

The difference between C and FORTRAN are as follows:

- C and FORTRAN are both programming languages. And both the languages are the oldest in the programming industry. One language developed in the 1950s and the other one developed in the 1970s.
- Both languages are commonly used for scientific computing. They both have their pros and cons. But both can be utilized for different purposes.
- The main C vs FORTRAN difference is that FORTRAN is design to handle large numerical problems. Whereas C (Take C programming assignment help to in-depth knowledge about this language) is more general purpose.
- FORTRAN is an interactive programming language. It was developed specifically for scientific computing needs.
- C is a general-purpose computer language that uses to write operating systems and commercial software.
- While C++ and C are procedural languages, FORTRAN is a mix of both procedural and object-oriented programming syntax.

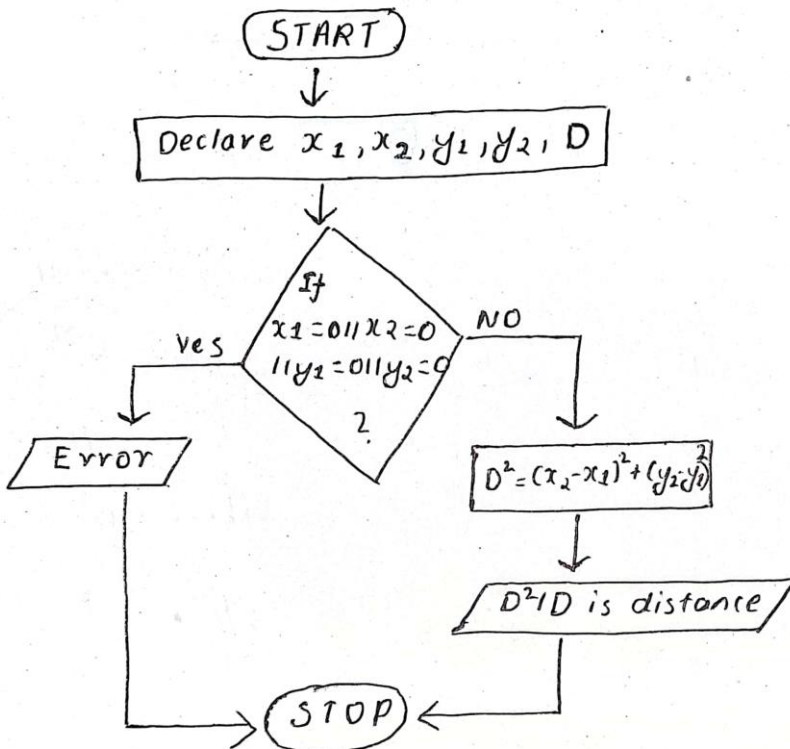
2. Write an algorithm and flowchart of the distance between two points (x_1, y_1) and (x_2, y_2) , governed by formula $D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$. Where, x_1, y_1, x_2, y_2 are given by user, but should not be zero. [5+5]

Answer

ALGORITHM

1. Start
2. Declare D, x_1, y_1, x_2, y_2
3. Read x_1, y_1, x_2, y_2 from user
4. If $x_1=0$ or $x_2=0$ or $y_1=0$ or $y_2=0$
Display error and go to 7
5. Let $D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$
6. Display D^2/D as distance
7. STOP

FLOWCHART



3. Write a syntax used in C programming language for the followings: [2*5]
a) Scanf() b) While c) struct d) if...else e) static

Answer

- a) Scanf():
Scanf(control string, arg1, arg2,);
- b) While:
while(condition) {
 statement(s);
}
- c) struct
struct structureName {
 dataType member1;
 dataType member2;
 ...
};
- d) if...else
if (test expression) {
 // run code if test expression is true
}
else {
 // run code if test expression is false
}
- e) static
static data_type var_name = var_value;

4. What are the significant meanings of ‘&’ and ‘*’ established in C programming? How can you differentiate between ‘called by value’ and ‘called by reference’ with example in C programming? [4+6]

answer

The "&" operator in C is the "address of" operator. It is used to obtain the memory address of a variable. For example, if you have an integer variable "x" and you want to know its memory address, you would use "&x".

The "*" operator in C is the "dereference" operator. It is used to obtain the value stored at a memory address. For example, if you have a pointer variable "p" that points to an integer and you want to know the value stored at that memory address, you would use "*p". This operator is also used in pointer declaration and function argument.

The following table lists the differences between the call-by-value and call-by-reference methods of parameter passing:

Call By Value	Call By Reference
While calling a function, we pass the values of variables to it. Such functions are known as “Call By Values”.	While calling a function, instead of passing the values of variables, we pass the address of variables (location of variables) to the function known as “Call By References.
In this method, the value of each variable in the calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function is copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them.
In call-by-values, we cannot alter the values of actual variables through function calls.	In call by reference, we can alter the values of variables through function calls.
Values of variables are passed by the Simple technique.	Pointer variables are necessary to define to store the address values of variables.
This method is preferred when we have to pass some small values that should not change.	This method is preferred when we have to pass a large amount of data to the function.

Example of call by value:

//swap by value example

```

#include<stdio.h>

void main()
{
    int a,b,temp;
    printf("Enter the values of a and b:\n");
    scanf("%d%d",&a,&b);
    printf("Before swapping : a= %d and b=%d\n",a,b);
    temp=a;
    a=b;
    b=temp;
    printf("After swapping : a= %d and b=%d",a,b);
}

```

OUTPUT

```

C:\Users\pande\Desktop\hello\swap\bin\Debug\swap.exe
Enter the values of a and b:
6
9
Before swapping : a= 6 and b=9
After swapping : a= 9 and b=6
Process returned 29 (0x1D)   execution time : 4.867 s
Press any key to continue.

```

Example of swap by reference:

```

//swap by reference example
//swap by reference example
#include <stdio.h>

void swap(int*,int*);

void main()
{
    int x,y;
    printf("Enter the values for x and y:\n");
    scanf("%d %d",&x,&y);
}

```



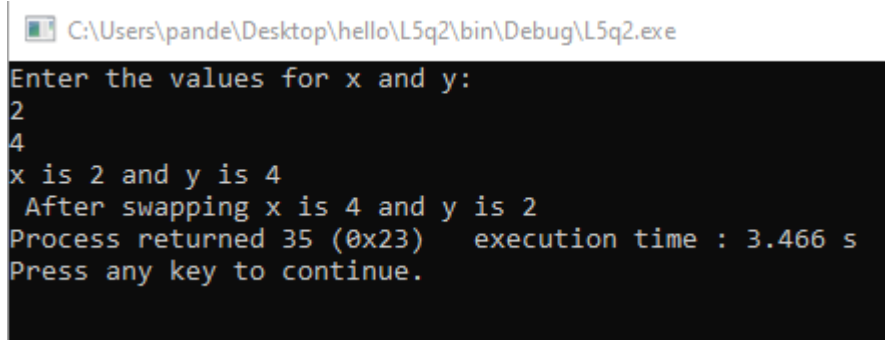
```

printf("x is %d and y is %d",x,y);
swap(&x,&y);
printf("\n After swapping x is %d and y is %d ",x,y);
}
void swap(int *a,int *b)
{
int temp;

temp=*a;
*a=*b;
*b=temp;
}

```

OUTPUT



```

C:\Users\pande\Desktop\hello\L5q2\bin\Debug\L5q2.exe
Enter the values for x and y:
2
4
x is 2 and y is 4
After swapping x is 4 and y is 2
Process returned 35 (0x23)   execution time : 3.466 s
Press any key to continue.

```

5. State with example, how switch () function differs from user defined function in computer programming language C. [4]

Answer

Switch case and user-defined functions are two different things. Switch is usually used when the user is given a specific cases to choose from. For ex: Displaying number of days in a month. To get the correct output for the above question, switch case is used. It has 12 cases which each cases pertaining to the 12 months of the year.

Case 1→January

Case 2→ February and so on...

When a user chooses a month number, the computer will search for that particular case and do the task specified to the case. This is just one simple example of the application of switch case.

User defined functions are used to increase modularity, readability and it also reduces the time consumed for debugging, testing etc. User defined functions is a part of the code which is quite effective when there 100s of lines of code. User defined function is a block of code called by a user

defined name inside which is a piece of code to complete a task. The code inside the function can be as long as needed and it can also contain other user defined functions.

The function can also contain a switch case inside it.

For ex: Let months () be a user defined function. As explained above the switch case for number of days in a month can be written inside the function months (). Hence when the function months() is called (instructed to execute) the switch case is prompted.

6. Explain the relation of array and pointer. What is meant by call by value and call by reference? Write a complete program that adds the corresponding elements of two matrices if the elements are positive, otherwise multiply the corresponding elements using the concept of passing array to the function and pointer.

Ans: An array is a data structure that holds a collection of similar type data. A pointer is a variable that holds the address of another variable. Arrays and pointers are intimately related in C. Arrays decay into pointers in certain contexts, and both can be used interchangeably for accessing and manipulating data. Arrays and pointers are similar in terms of syntax, especially when it comes to accessing elements.

When you use an index to access an array element, it is equivalent to using pointer arithmetic to access the same element. For example: `int x = arr[2];` // This is equivalent to: `int y = *(arr + 2);`

Call by value and call by reference are terms used to describe two different ways of passing arguments to functions in programming languages. In call by value, a copy of the argument's value is passed to the function. This means that any changes made to the argument within the function do not affect the original value outside the function. The function receives its own copy of the argument and works with that copy independently. In call by reference, a reference or memory address of the argument is passed to the function. This means that changes made to the argument within the function directly affect the original value outside the function. The function works with the actual variable in memory, not a copy

```
#include <stdio.h> int main() {  
  
    int r, c, a[100][100], b[100][100], sum[100][100], i, j; printf("Enter the  
number of rows (between 1 and 100): "); scanf("%d", &r);  
  
    printf("Enter the number of columns (between 1 and 100): "); scanf("%d", &c);  
  
    printf("\nEnter elements of 1st matrix:\n"); for (i = 0; i  
< r; ++i)  
        for (j = 0; j < c; ++j) {  
            printf("Enter element a%d%d: ", i + 1, j + 1);  
            scanf("%d", &a[i][j]);  
        }  
  
    printf("Enter elements of 2nd matrix:\n"); for (i = 0; i  
< r; ++i)
```

```

for (j = 0; j < c; ++j) {
    printf("Enter element b%d%d: ", i + 1, j + 1);
    scanf("%d", &b[i][j]);
}

// adding two matrices
for (i = 0; i < r; ++i)    for (j =
0; j < c; ++j) {
    sum[i][j] = a[i][j] + b[i][j];
}

// printing the result
printf("\nSum of two matrices: \n"); for (i = 0;
i < r; ++i)    for (j = 0; j < c; ++j) {    printf("%d
", sum[i][j]);
    if (j == c - 1) {
        printf("\n\n");
    }
}

return 0;
}

```

```

-----
Process exited after 42.12 seconds with return value 0
Press any key to continue . . .

```

7. Define the structures in C programming? Explain nested structure with suitable example.

Write a program using structure and passing the structure to the function that returns the result to convert the date (in format YY/MM/DD) in B.S to A.D.

Ans: In C programming, a structure is a composite data type that groups together variables of different data types under a single name. It allows you to define a custom data structure that can hold related pieces of data. Structures are used to represent records, objects, or entities in your program.

Here's the syntax:

```

struct StructureName { DataType
member1;

```

```
DataType member2;
```

```
};
```

A nested structure in C is a structure that is defined inside another structure. This allows you to create more complex data structures by combining multiple types of data. Nested structures are particularly useful when you need to model hierarchical relationships between data elements. Example for the nested structure loop:

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

```
struct Point {    int x;
```

```
int y;
```

```
};
```

```
int main() {
```

```
    int numPoints;
```

```
    printf("Enter the number of points: ");
```

```
    scanf("%d", &numPoints);
```

```
    struct Point points[numPoints];    for (int i =  
0; i < numPoints; i++) {
```

```
        printf("Enter coordinates for Point %d (x y): ", i + 1);    scanf("%d  
%d", &points[i].x, &points[i].y);
```

```
    }
```

```
    printf("Entered points:\n");    for (int i =  
0; i < numPoints; i++) {
```

```
        printf("Point %d: (%d, %d)\n", i + 1, points[i].x, points[i].y);
```

```
    }
```

```
    return 0;
```

```
}
```

OUTPUT:



8.Explain the I/O operations on file with suitable example and state the typical error situations during I/O operations in file. How the contents in the file can be randomly accessed?

Ans: Input/Output (I/O) operations on files in programming involve reading from and writing to files. Files are essential for persistent data storage and retrieval beyond the program's execution.

File I/O operations generally involve the following steps:

- **Opening a File:** To perform I/O operations on a file, you need to open it first. This is done using functions like **fopen()** in C. The **fopen()** function returns a file pointer, which is used to refer to the opened file.
- **Reading from a File:** You can read data from a file using functions like **fscanf()** or **fgets()** in C. These functions allow you to read data from the file and store it in variables.
- **Writing to a File:** Writing data to a file is done using functions like **fprintf()** or **fputs()** in C. These functions allow you to write data to the file from variables.
- **Closing a File:** After you're done with the file, it's important to close it using the **fclose()** function. This ensures that any buffered data is flushed, and resources are released.

Typical Error Situations during File I/O Operations:

- **File Not Found:** Trying to open a file that doesn't exist can result in a "File not found" error. Always check the return value of **fopen()** for **NULL** to detect such errors.
- **Permission Issues:** If the program doesn't have the necessary permissions to access the file (e.g., read or write permissions), file I/O operations can fail.
- **Buffer Overflows:** Reading or writing more data than the buffer size can lead to buffer overflows, which can corrupt memory and crash the program.
- **File Handle Exhaustion:** Operating systems often limit the number of open files. Failing to close files properly can lead to running out of available file handles.
- **Disk Full:** Writing to a disk with insufficient space can lead to "Disk full" errors.
- **Improper Formatting:** When reading/writing formatted data, incorrect formatting can lead to data misinterpretation or errors.
- **File Locking:** In multi-process or multi-threaded environments, improper file locking can result in data inconsistencies.

To handle these errors, we should check return values of I/O functions and handle them gracefully with appropriate error messages or exception handling mechanisms.

And in C, the standard library provides functions that enable random access to file contents:

- **fseek():** This function is used to set the file position indicator to a specific position within the file. It takes the file pointer, an offset value, and a position indicator (such as **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**).
- **ftell():** This function returns the current file position indicator's offset from the beginning of the file.
- **rewind():** This function sets the file position indicator to the beginning of the file, equivalent to calling **fseek(file, 0, SEEK_SET)**.

With these functions, we can move the file pointer to a desired position and then use read or write operations to interact with the file contents at that position.

9. What are FORTAN's constants, variables and library functions? Explain. Write a program in FORTAN to take a position and a number and insert this number on this position inside an array containing n elements.

Ans: Fortran (short for "Formula Translation") is a programming language that has evolved over several versions and standards. It is commonly used in scientific and engineering applications. Here's a brief overview of constants, variables, and library functions in Fortran:

Constants in Fortran: Fortran supports several types of constants:

- **Numeric Constants:** These include integer constants (e.g., 42), real constants (e.g., 3.14), and complex constants (e.g., (1.0, 2.0)).
- **Logical Constants:** These are the logical values TRUE and FALSE.
- **Character Constants:** These are sequences of characters enclosed in single quotes (e.g., 'Hello, World!').

Variables in Fortran: In Fortran, variables are used to store data. Variable names can consist of up to 31 alphanumeric characters (starting with a letter) and can include underscores.

Variable names are not case-sensitive in Fortran.

Library Functions in Fortran: Fortran has a set of built-in intrinsic functions that provide various mathematical, string manipulation, and other operations. Here are a few examples:

- **Mathematical Functions:** Fortran provides various mathematical functions like SIN, COS, EXP, SQRT, etc.
- **String Manipulation:** Fortran allows string manipulation through functions like LEN, TRIM, INDEX, SCAN, etc.
- **Input/Output Functions:** Fortran includes I/O functions such as READ, WRITE, OPEN, CLOSE, etc., for reading from and writing to files.
- **Array Functions:** Fortran provides array-related functions like SUM, MAXVAL, MINVAL, etc., to perform operations on arrays.
- **Date and Time Functions:** Some Fortran compilers offer date and time functions to work with dates and times.

```
program insert
  implicit none
  integer :: n, pos,
  num, i
  integer, allocatable :: arr(:)
  print *,
  "Enter the size of the array:"
  read *, n
  allocate(arr(n))

  print *, "Enter the elements of the array:"
  do i = 1, n
    read *,
    arr(i)
  end do
  print *, "Enter the position and the number to
  insert:"
  read *, pos, num
  if (pos < 1 .or. pos > n) then
    print *,
    "Invalid position"

    stop
  end if

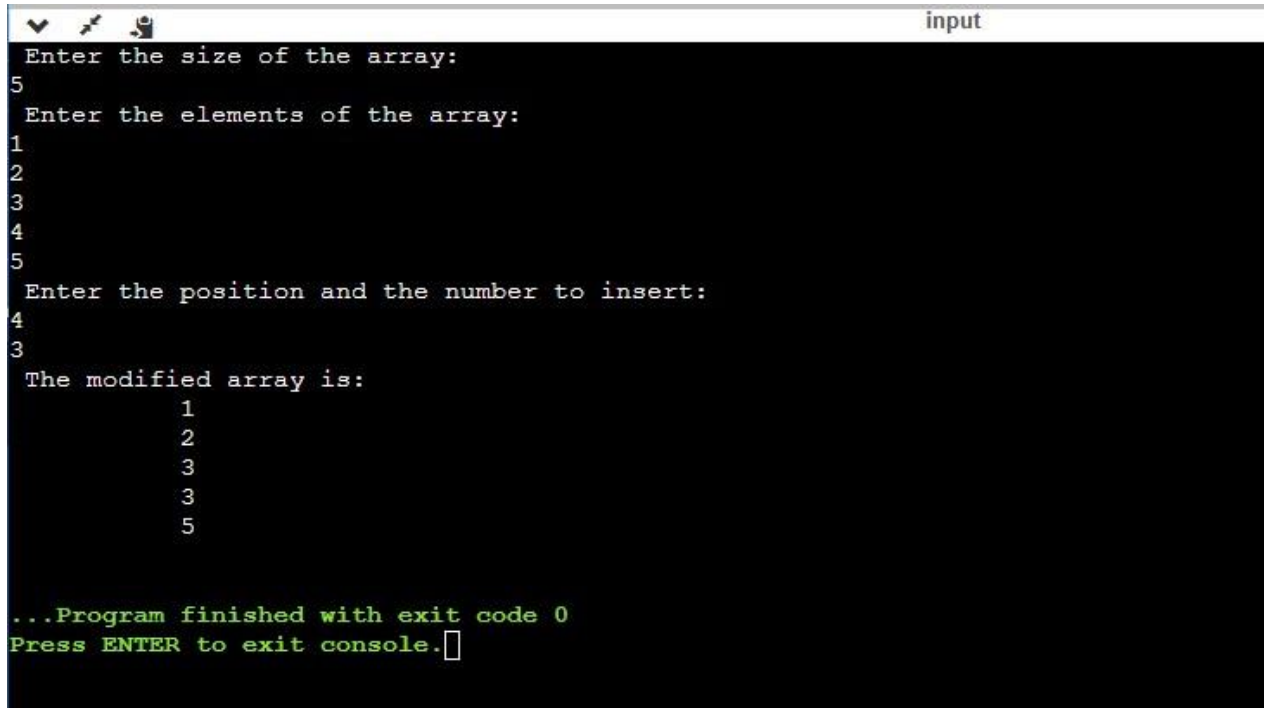
  arr(pos) = num ! insert the number at the position
  print *, "The
  modified array is:"

  do i = 1, n
```

```
    print *, arr(i) end do
deallocate(arr)

end program insert
```

OUTPUT:



```
input
Enter the size of the array:
5
Enter the elements of the array:
1
2
3
4
5
Enter the position and the number to insert:
4
3
The modified array is:
    1
    2
    3
    3
    5

...Program finished with exit code 0
Press ENTER to exit console.□
```

2080 baisakh Solutions

1.a) How is an application software different from system software? Explain with examples.

Ans: Application software refers to programs designed to perform specific tasks or functions for end-users. It is created to address needs or provide services directly to users. Application software is what most people interact with daily. It's often developed to serve a variety of purposes, from productivity and entertainment to communication and creativity. Examples are MS words, Adobe Photoshop, Google Chrome, Spotify, etc.

System software, on the other hand, is responsible for managing and controlling the computer hardware and providing a platform for running application software. It works behind the scenes to ensure that the computer's hardware components function together effectively and efficiently. System software acts as an intermediary between hardware and application software. Examples are Operating System, Compilers, antivirus software, etc.

Application software is designed to perform specific tasks for end-users, while system software manages and facilitates the interaction between hardware and application software. Application software provides functionality and features directly to users, while system software ensures the overall functionality and stability of the computer system.

b) How is code in source file is converted to the executable file? Explain with key steps involved.

Ans: The process of converting human-readable source code into an executable file that a computer can understand and execute involves several key steps. This process is often referred to as the compilation and linking process. Here are the main steps involved:

- 1. Writing the Source Code:** The process begins with a programmer writing the source code using a programming language like C, C++, Java, or Python. The source code contains instructions that outline the logic and behaviour of the program.
- 2. Preprocessing:** Before the code is compiled, a preprocessor performs certain tasks like including header files, macro expansions, and conditional compilation. It prepares the code for compilation by making necessary replacements and modifications.

3. Compilation:

The source code is then fed into a compiler, which translates the human-readable code into an intermediate form called object code or machine code. This step involves syntax analysis, semantic analysis, and code generation. The compiler checks for syntax errors and translates the code into a lower-level representation that the computer's CPU can understand.

- 4. Object Files Generation:** The output of the compilation step is one or more object files, which contain the translated code for individual source files. These object files typically have a ".o" or ".obj" extension.
- 5. Linking:** If the program consists of multiple source files or external libraries, the linker comes into play. The linker combines these object files along with the necessary libraries to create an executable file. It resolves references between different parts of the program, ensuring that functions and variables are linked correctly.

- 6. Symbol Resolution:** During the linking phase, the linker resolves symbols, which are references to functions or variables. It ensures that each symbol is correctly connected to its implementation, either within the program's own code or in external libraries.
- 7. Relocation:** The linker also performs relocation, adjusting memory addresses within the object code to match the actual memory addresses where the program will be loaded and executed.
- 8. Creating Executable:** After resolving symbols and performing relocation, the linker generates the final executable file. This file contains machine code that can be directly loaded into memory and executed by the computer's CPU.
- 9. Loading and Execution:** When the user runs the executable file, the operating system's loader brings the program into memory, allocates necessary resources, and starts the execution by jumping to the program's entry point (often the **main** function in C-like languages). The CPU then interprets and executes the machine code instructions.

Thus, this is how code in source file is converted to the executable file.

2.a) How is constant different than a variable?

Ans: A constant is a value that remains unchanged throughout the entire execution of a program. It's a fixed, unalterable piece of data that is assigned a value at the time of declaration and cannot be modified thereafter. Constants are used to represent values that are known and unchanging, such as mathematical constants or configuration values.

Key characteristics of constants:

- Assigned a value only once, typically at the time of declaration.
- Cannot be modified or updated during the program's execution.
- Used for values that remain the same throughout the program's execution. A variable, on the other hand, is a named storage location in a program's memory that holds data that can vary or change during the program's execution. Variables allow programmers to store, manipulate, and update data dynamically as the program runs. They are used to store information that might change over time, such as user input, calculations, and intermediate results.

Key characteristics of variables:

- Can be assigned a value initially and can be updated with new values during the program's execution.
- Can hold different values at different points in time.
- Used to store data that can change or needs to be manipulated.

Constants are unchanging values that are assigned once and remain the same throughout a program's execution, while variables are named memory locations that can hold different values over time.

b) Write a C program to read an integer 'd' from the user. If 'd' is the radius of a circular ground in meters, then this program should calculate and display the area of the circle in square meter.

Ans:

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

```
int main() {    int d;
    float radius, area;

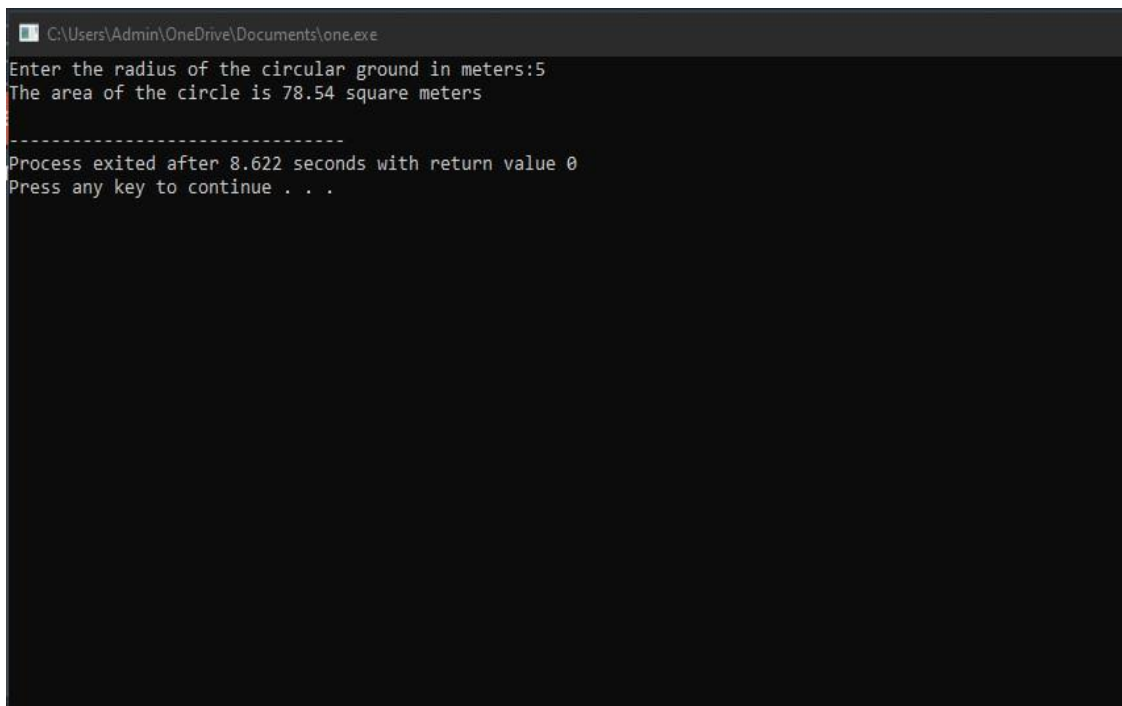
    printf("Enter the radius of the circular ground in meters: ");    scanf("%d", &d);

    radius = (float)d;
    area = 3.14159 * radius * radius;

    printf("The area of the circle is %.2f square meters\n", area);

    return 0;
}
```

OUTPUT:



```
C:\Users\Admin\OneDrive\Documents\one.exe
Enter the radius of the circular ground in meters:5
The area of the circle is 78.54 square meters

-----
Process exited after 8.622 seconds with return value 0
Press any key to continue . . .
```

Q-N (Q.N-3): What are formatted input/output functions? Explain each in detail with examples.

Ans: A function is a self-contained block of code that performs a specific task. Functions. There are two main types of functions when it comes to input/output operation:

1. f

1. Formatted input/output function

2. Unformatted I/O function.

1. Formatted input/output function:-

Those data types function which deals with formatted data, which means the data is presented in a specific way, such that, ^{as} numbers with decimal places, strings with specific alignments, etc. These types of I/O functions can help to display the output to the user in different formats using the format specifiers. These I/O supports all data types like int, float, char and many more.

Examples:-

• a. printf: It is used in C program to display formatted output on the console screen. It is a pre-defined function that is already declared in the stdio.h (header file).

Syntax: printf("Format specifier", var1, var2, ..., varN);

1. f. scanf : It is used to read formatted input from the console. It is used reading or taking any value from keyboard by the user, these values can be any type like float, int, char, string, etc. The function is declared in `stdio.h` & it is pre-defined junction. In `scanf` we use `&` (address of operator) to store the variable value on the memory location of that variable.

Syntax: `scanf("Format specifier" &Var1, &Var2, &Var3, ..., &VarN);`

2. Unformatted I/O function:

It deals with binary data which means data is not formatted or converted into human readable form. These function, are used to read or write raw data, such as arrays of bytes or binary structures

=> Examples of unformatted output function:

a. fwrite : It is used to write binary data to a file.

Syntax: `fwrite("data", size of (int), 5, file);`

=> Example of unformatted input funcⁿ:

b. fread : It is used to read binary data from a file.

(Q.N-4) (a): Differentiate between library function and user defined functions. Provide relevant examples.

Key for difference	Library function	User defined function.
Definition ² Origin	Library function are pre-defined functions provided by the programming language or external libraries.	1. User defined function are the functions that a programmer create and define within your code.
Availability	2. Library functions are readily available & donot require explicit implementation.	2. These functions need to be defined by the programmer.
Customization	3. It can provide general purpose-functionality.	3. It can be tailored to specific requirements.
Modularity.	4. They are already organised & packaged for use.	4. These functions promote code modularity and organization, making the code easier to manage.

Q. N-4 (b) :- What is function prototype?
Write a C program to find sum of first 10 natural numbers using recursion.

Ans:- A function prototype is a declaration of a function that specifies the function's name & type return type and the types of its parameters.

⇒ CODE for sum of first 10 natural number is :-

```
#include <stdio.h>
int sum(int n) {
    if (n == 0)
        return 0;
    else
        return n + sum(n-1);
}
```

```
int main() {
    int n = 10;
    int sum =
    int s = sum(n);
    printf("Sum of first 10 natural numbers is: %d", n, sum);
    return 0;
}
```

Q. Q.N-5(a): WAP to read marks of 48 students in a class and display second highest

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    float arr[48], temp;
    printf ("Enter the marks of 48 students");
    {
        scanf ("%f", &arr[i]);
    }
    for (int i=0; i<48; i++)
    {
        for (int j=i+1; j<48; j++)
        {
            if (arr[i] < arr[j])
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
    printf ("Second largest number is %.2f, arr[i]");
}
```

(Q.N-6) What do you mean by array of structure? WAP to create a structure named "student" having roll members as roll, name, address and marks. Use this structure to read the information of 48 students in a class and display the information of only those students whose marks is between 50 & 70.

Ans: An array of structure is a data structure in programming where multiple instance of a structure are stored in contiguous memory locations as an elements of array.

=> CODE:

```
#include <stdio.h>
#include <string.h>
struct student
{
    int roll;
    char name [20];
    char address [50];
    int marks;
};
int main()
{
    const int N = 48;
    struct student S [48];
    for (int i = 0; i < 48; i++)
    {
```



```
printf ("Enter the details for student  
%d:\n", i+1);
```

```
printf  
scanf ("%d, %s, %s, %d", &S[i].roll,  
S[i].name, &S[i].address,  
&S[i].marks);
```

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

```
printf ("Display the marks between  
50 & 70 :\n");
```

```
if (S[i].marks >= 50 && S[i].marks <= 70  
{
```

```
printf ("%d, %s, %s, %d", S[i].roll,  
S[i].name, S[i].address,  
S[i].marks);  
printf ("\n");
```

```
}  
}  
return 0;  
}
```

program prime number

integer n, i

write (*, *), n

do

if (mod(n, i).eq.0) then

goto 900

end if

enddo

900 if (i, 2.

write (*, *), 'The number is prime'

(Q.N-7) What are pointer arithmetic? Explain with example. Write in C program that uses pointer to read $m \times n$ matrix from user. Pass it to function that finds the transpose of the matrix.

Ans: Pointer arithmetic is one of the powerful set of valid arithmetic operation that can be performed on pointer. Some of these are:

a. C pointer increment:

Increment a pointer in C simply means to increase the pointer value step by step to point to next location.

Logic: $1061C$:

Next Location = Current Location + $i * \text{Size of (data type)}$.

b. C pointer decrement:

Decrementing a pointer in C simply means to decrease the pointer value step by step to point the previous location.

logic: $\text{Next location} = \text{Current location} - i * \text{Size of (data type)}$.

c. C pointer addition:

Addition in pointer simply means add a value to the pointer value

```
#include <stdio.h>
void main()
{
    int n = 10;
    int *a;
    a = &n;
    printf ("current location = %x \n", a);
    a = a + 4;
    printf ("next address = %x \n", a);
}
```

d. C pointer Substraction.

Substraction from a pointer in C simply means to substract a value from the pointer value.



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

```
void transposematrix (int *matrix,
int m, int n) {
```

```
int transpose [n][m];
```

```
for (int i=0; i<n; i++)
{
```

```
for (int j=0; j<m; j++)
{
```

```
transpose [i][j] = *(matrix + j * n + i);
```

```
}
```

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

```
for (int i=0; i<n; i++)
{
```

```
for (int j=0; j<m; j++)
{
```

```
printf ("%d\t", transpose [i][j]);
```

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

```
}
```

Q.N-8 (a) How is a binary file differ from text file? Write the syntax & use of fseek() and rewind() functions in C.

⇒ Binary files and text files are two different types of files used to store data in a computer system. They differ in their internal structure, content representation and how they are processed by software. Here are the main differences between binary & text files.

1. CONTENT REPRESENTATION:-

Binary	Text
1. Binary file contain data in a format that is not directly human-readable. They store information as sequences of bytes, which can represent a wide range of data types, including numbers, characters, images, audio and more. Binary files are meant to be interpreted by humans without specialised software.	Text files store data in a human readable form. They primarily contain characters from a specific character encoding (ASCII or unicode) that represent text, numbers and symbols. Text files can be opened and read using simple text editors.

2. INTERNAL STRUCTURE:

Binary	Text
Binary files have a structure that depends on the specific format and purpose of file. The bytes in a binary file can be organised in a various way to represent complex data structures, including headers, metadata & actual data.	Text files have a straightforward structure. They consist of sequences of characters, typically encoded in a way that maps each character to a numerical value.

=> `fseek()` & `rewind()` function are used for file handling to navigate within a file. They are part of `<stdio.h>` and are used with 'FILE' pointer to manipulate the current position indicator within a file.

1. `fseek()` function:

Syntax: `int fseek (FILE * stream, long offset, int origin);`

USE: The '`fseek()`' function is used to move the file pointer to a specified position within a file.

2. rewind() function:

Syntax: `void rewind (FILE *stream);`

USE:

It is a simpler way to move the file pointer to the beginning of the file.
It is equivalent to using `fseek (file, 0, SEEK_SET)`

Q.N-8(C):

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

```
struct movie  
{  
    char name [100];  
    int year of release;  
    char language [50];  
};
```

```
int main ()  
{  
    FILE * datafile = fopen ("english movies.  
dat", "wb");  
    if (datafile == NULL)  
    {  
        printf ("File can't be opened\n");  
        return 1;  
    }  
}
```

e. (Contd...)

```

struct Movie movies',
for (int i=0; i<3; i++)
{

```

```

printf ("Enter year of name movie %d:", i+1);
scanf ("%d", &movie.name);

```

```

printf ("Enter year of release of movie
%d:", i+1);
scanf ("%d", &movie.year_of_release);

```

```

printf ("Enter language of movie %d:",
i+1);
scanf ("%s", movie.language);

```

```

if (strcmp (movie.language, "english")
== 0)

```

```

{
fwrite (&movie, sizeof (struct Movie),
1, datafile);
}

```

```

}
fclose (datafile);
printf ("Movie data saved successfully
\n");

```

```

return 0;
}

```


(Q.N-9) What is the use of FORMAT statement? Explain I & F format with example. WAP in FORTRAN to read a number from user & check whether it is a prime number or not.

Ans: In FORTRAN, the FORMAT statement is used to control the input and output formatting of data when reading from or writing to files. It specifies how data should be formatted and how it should be presented to the user or stored in a file.

→ Two commonly used descriptors are 'I' & 'F'

a. 'I' Format descriptor:-

The 'I' format descriptor is used for formatted I/O of integer value. It is used to specify the width & position of integer data within a formatted I/O statement.

Ex:

PROGRAM Format Example.

Integer :: num

WRITE (*, "(I5)") 123

READ (*, "(I3)") num

WRITE (*, "(A,T3)") "Number:", num

END PROGRAM.

6. 'F' format descriptor: The "F" format is used for the formatted I/O of floating-point values. It is used to specify the width, position, and decimal place of floating point data within formatted I/O statement.

Example:

```
PROGRAM Format Example.
REAL::value.
WRITE (*, "(F8.2)") 123.456.
READ (*, "(F6.2)") value
WRITE (*, "(A, F6.2)") "value: ", value
END PROGRAM
```

```
3) PROGRAM PRIME CHECK
INTEGER:: num, i, factor
LOGICAL:: is_prime.

WRITE (*, *) "Enter a number:"
READ (*, *) num.
```

```
is_prime = .TRUE.
IF (num <= 1) Then
    is_prime = .FALSE.
```

```
ELSE
    DO i=2, num-1
        IF (MOD(num, i) == 0) THEN
            is_prime = .FALSE.
```

```
END IF  
END DO  
END IF
```

```
IF (is_prime) THEN  
  WRITE (*, *) num, "is a prime number."  
ELSE  
  WRITE (*, *) num, "is not a prime number."  
END IF  
END PROGRAM
```

(Q.N-10) Write short notes on.

a. While & do-while statement:

The 'while' statement is used to create a loop that contains to execute a block of code as long as a specified condition is true. The basic syntax is:-
while (cond..)

In this structure, the condⁿ is checked before the execution of the code block. If the condition is true, the code block is executed & the condⁿ is checked again. This process continues until the condⁿ becomes false, at which point the loop terminates, and the program continues with the next statement after the loop.

Ex:-

```
Count = 0  
while (Count < 5)
```