Program:16 /*program to add,subtract,multiply and divide two integers using user defined type function with return type.*/

```
#include<stdio.h>
#include<conio.h>
void sum(int x, int y);
void sub(int x, int y);
void mult(int x, int y);
void div(int x, int y);
void main()
{
int a,b;
printf("enter two numbers:");
scanf("%d %d",&a,&b);
sum(a,b);
sub(a,b);
mult(a,b);
div(a,b);
getch();
}
void sum(int x,int y)
{
printf("sum=%d\n",x+y);
}
void sub(int x,int y)
{
printf("difference=%d\n",x-y);
}
void mult(int x,int y)
{
printf("product=%d\n",x*y);
}
void div(int x,int y)
{
printf("quotient=%f\n",(float)x/y);
}
```

Program 17: //Factorial Program using loop

```
#include<stdio.h>
int main()
{
 int i,fact=1,number;
 printf("Enter a number: ");
  scanf("%d",&number);
    for(i=1;i<=number;i++){
      fact=fact*i;
  }
  printf("Factorial of %d is: %d",number,fact);
return 0;
}
//Factorial Program using recursion in C
#include<stdio.h>
long factorial(int n)
```

```
{    if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
void main()
{
  int number;
  long fact;
  printf("Enter a number: ");
  scanf("%d", &number);
  fact = factorial(number);
  printf("Factorial of %d is %ld\n", number, fact);
  return 0;
}
```

//C Program to Find Factorial of a Number using Functions

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

```
#include<math.h>
int main()
{
    printf("Enter a Number to Find Factorial: ");
    printf("\nFactorial of a Given Number is: %d ",fact());
    return 0;
}
int fact()
{
    int i,fact=1,n;
    scanf("%d",&n);
    for(i=1; i<=n; i++)
    {
        fact=fact*i;
    }
    return fact;
}
```

Program 18: Difference between call by value and call by reference in c

.

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Call by Value in C

Calling a function by value will cause the program to copy the contents of an object passed into a function. To implement this in C, a function declaration has the following form: [return type] functionName([type][parameter name],...).

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>

void swap(int x, int y){

    int temp = x;

    x = y;

    y = temp;

}

int main(){

    int x = 10;

    int y = 11;

    printf("Values before swap: x = %d, y = %d\n", x,y);

    swap(x,y);

    printf("Values after swap: x = %d, y = %d", x,y);

}
```

OUTPUT

```
Values before swap: x = 10, y = 11

Values after swap: x = 10, y = 11
```

We can observe that even when we change the content of x and y in the scope of the swap function, these changes do not reflect on x and y variables defined in the scope of main. This is because we call swap() by value and it will get separate memory for x and y so the changes made in swap() will not reflect in main().

Call by Reference in C

Calling a function by reference will give function parameter the address of original parameter due to which they will point to same memory location and any changes made in the function parameter will also reflect in original parameters. To implement this in C, a function declaration has the following form: [return type] functionName([type]* [parameter name],...).

```
#include <stdio.h>

void swap(int *x, int *y){

    int temp = *x;

    *x = *y;

    *y = temp;

}
```

```
int main(){
    int x = 10;
    int y = 11;
    printf("Values before swap: x = %d, y = %d\n", x,y);
    swap(&x,&y);
    printf("Values after swap: x = %d, y = %d", x,y);
}
```

Output:

```
Values before swap: x = 10, y = 11
Values after swap: x = 11, y = 10
```

We can observe in function parameters instead of using int x,int y we used int *x,int *y and in function call instead of giving x,y, we give &x,&y this methodology is call by reference as we used pointers as function parameter which will get original parameters' address instead of their value. & operator is used to give address of the variables and * is used to access the memory location that pointer is pointing. As the function variable is pointing to the same memory location as the original parameters, the changes made in swap() reflect in main() which we can see in the above output.

When to Use Call by Value and Call by Reference in C?

Copying is expensive, and we have to use our resources wisely. Imagine copying a large object like an array with over a million elements only to enumerate the values inside the array, doing so will result in a waste of time and memory. Time is valuable and we can omit to copy when:

We intend to read state information about an object, or

Allow a function to modify the state of our object.

However, when we do not intend our function to alter the state of our object outside of our function, copying prevents us from making unintentional mistakes and introduce bugs. Now we know when to use call by value and call by reference in C.

Now we will discuss the advantages and disadvantages of call by value and call by reference in C.

Advantages of Using Call by Value Method

Guarantees that changes that alter the behavior of a parameter stay within its scope and do not affect the value of an object passed into the function

Reduce the chance of introducing subtle bugs which can be difficult to monitor.

Passing by value removes the possible side effects of a function which makes your program easier to maintain and reason with.

Advantages of Using Call by Reference Method

Calling a function by reference does not incur performance penalties that copying would require. Likewise, it does not duplicate the memory necessary to access the content of an object that resides in our program.

Allows a function to update the value of an object that is passed into it.

Allows you to pass functions as references through a technique called function pointers which may alter the behavior of a function. Likewise, lambda expressions may also be passed inside a function. Both enable function composition which has neat theoretical properties.

Disadvantages of Using Call by Value Method

Incurs performance penalty when copying large objects.

Requires to reallocate memory with the same size as the object passed into the function.

Disadvantages of Using Call by Reference Method

For every function that shares with the same object, your responsibility of tracking each change also expands.

Making sure that the object does not die out abruptly is a serious issue about calling a function by reference. This is especially true in the context of a multithreaded application.

Program:19 Write a program to transpose a matrix.

The transpose of a matrix is a new matrix that is obtained by exchanging the rows and columns.

In this program, the user is asked to enter the number of rows r and columns c. Their values should be less than 10 in this program.

Then, the user is asked to enter the elements of the matrix (of order r*c).

The program below then computes the transpose of the matrix and prints it on the screen

```
#include <stdio.h>

int main() {

  int a[10][10], transpose[10][10], r, c;

  printf("Enter rows and columns: ");

  scanf("%d %d", &r, &c);

  // asssigning elements to the matrix

  printf("\nEnter matrix elements:\n");

  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    printf("Enter element a%d%d: ", i + 1, j + 1);
    scanf("%d", &a[i][j]);
  }
  // printing the matrix a[][]
  printf("\nEntered matrix: \n");
  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    printf("%d  ", a[i][j]);
    if (j == c - 1)
    printf("\n");
  }
  // computing the transpose
  for (int i = 0; i < r; ++i)
  for (int j = 0; j < c; ++j) {
    transpose[j][i] = a[i][j];
  }
  // printing the transpose
```

```
  printf("\nTranspose of the matrix:\n");
  for (int i = 0; i < c; ++i)
  for (int j = 0; j < r; ++j) {
    printf("%d  ", transpose[i][j]);
    if (j == r - 1)
    printf("\n");
  }
  return 0;
}
```

Run Code

Output

```
Enter rows and columns: 2
3
Enter matrix elements:
Enter element a11: 1
Enter element a12: 4
Enter element a13: 0
Enter element a21: -5
Enter element a22: 2

Enter element a23: 7

Entered matrix:

1  4  0

-5  2  7

Transpose of the matrix:

1  -5

4  2

0  7
```

Program : 20 Matrix Multiplication in C

To multiply any two matrices in C programming, first ask the user to enter any two matrices, then start multiplying the given two matrices, and store the multiplication result one by one inside any variable, say sum. Store the value of sum in the third matrix (one by one as its element), say mat3, as shown in the program given here.

The question is, "Write a program in C that multiplies two given matrices." The answer to this question is given below. This C program asks the user to enter any two 3*3 matrix elements and multiply them to form a new matrix that is the multiplication result of the two given 3*3 matrices. Here, "3*3 matrix" means a matrix that has 3 rows and 3 columns:

```
#include<stdio.h>

#include<conio.h>

int main()

{

   int mat1[3][3], mat2[3][3], mat3[3][3], sum=0, i, j, k;

   printf("Enter first 3*3 matrix element: ");
```

```
for(i=0; i<3; i++)
{
    for(j=0; j<3; j++)
        scanf("%d", &mat1[i][j]);
}
printf("Enter second 3*3 matrix element: ");
for(i=0; i<3; i++)
{
    for(j=0; j<3; j++)
        scanf("%d", &mat2[i][j]);
}
printf("\nMultiplying two matrices...");
for(i=0; i<3; i++)
{
    for(j=0; j<3; j++)
    {
        sum=0;
        for(k=0; k<3; k++)
```

```
            sum = sum + mat1[i][k] * mat2[k][j];
         mat3[i][j] = sum;
      }
   }
   printf("\nMultiplication result of the two given Matrix is: \n");
   for(i=0; i<3; i++)
   {
      for(j=0; j<3; j++)
         printf("%d\t", mat3[i][j]);
      printf("\n");
   }
   getch();
   return 0;
}

// C Program to Multiply Two Matrix
#include <stdio.h>
```

```
int main()
{
    int A[3][3]={{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};
    int B[3][3]={{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};
    int i, j, k, C[3][3];
    printf("Matrix A :--> \n");
    for(i = 0;i < 3; i++)
    {
        for( j = 0; j < 3; j++)
        {
            printf("%d  ", A[i][j]);
        }
        printf("\n");
    }
    printf("\n\nMatrix B :--> \n");
    for(i = 0;i < 3; i++)
    {
        for( j = 0; j < 3; j++)
```

```
     {
          printf("%d  ", B[i][j]);
     }
     printf("\n");
}
// Multiplying Matrix
for(i =0 ;i < 3; i++)
{
     for(j = 0;j < 3; j++)
     {
       C[i][j]=0;
          for(k = 0; k < 3; k++)
          {
               C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
          }
     }
}
printf("\n\nResultant Matrix :--> \n");
```

```
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%d  ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
Matrix A :-->
1  1  1
2  2  2
3  3  3

Matrix B :-->
1  1  1
```

```
2  2  2

3  3  3

Resultant Matrix :-->

6  6  6

12  12  12

18  18  18
```