SOP-UNIT2

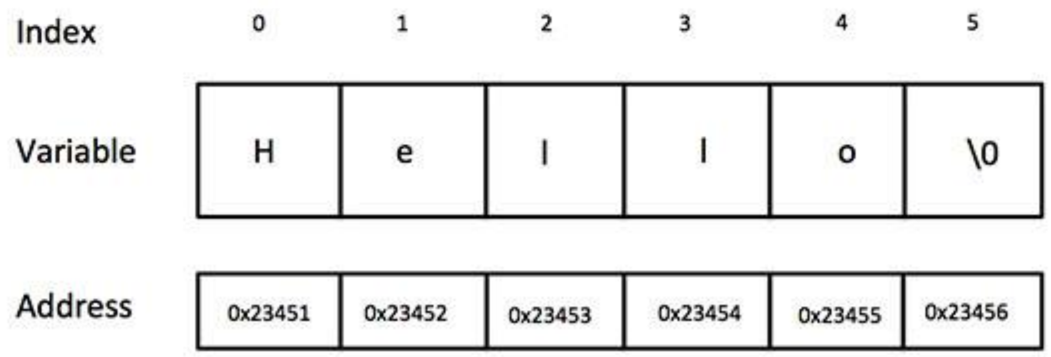
STRINGS

What are Strings?

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings. Many languages internally treat strings as character arrays, but somehow conceal this fact from the programmer. Character arrays or strings are used by programming languages to manipulate text, such as words and sentences. A string constant is a one-dimensional array of characters terminated by a null (’\0’). For example,

char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;

Each character in the array occupies 1 byte of memory and the last character is always ’\0’. What character is this? It looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special. ’\0’ is called null character. Note that ’\0’ and ’0’ are not same. ASCII value of ’\0’ is 0, whereas ASCII value of ’0’ is 48. Figure 15.1 shows the way a character array is stored in memory Note that the elements of the character array are stored in contiguous memory locations. The terminating null (’\0’) is important, because it is the only way the functions that work with a string can know where the string ends. In fact, a string not terminated by a ’\0’ is not really a string, but merely a collection of characters.



Also you can initialize array simply by:

```
char first_name[15] = "ANTHONY";
char string2 [ ] = "world";
```

In what way are character arrays different from numeric arrays? Can elements in a character array be accessed in the same way as the elements of a numeric array? Do I need to take any special care of '\0'? Why numeric arrays don't end with a '\0'? Declaring strings is okay, but how do I manipulate them? Questions galore!! Well, let us settle some of these issues right away with the help of sample programs.

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
```

String Input

The following sections will describe the methods of taking input from the user.

Using %s control string with scanf() Strings may be read by using the %s conversion with the function scanf() but there are some irksome restrictions. The fi rst is that scanf() only recognizes a sequence of characters delimited by white space characters as an external string. The second is that it is the programmer's responsibility to ensure that there is enough space to receive and store the incoming string along with the terminating null which is automatically generated and stored by scanf() as part of the %s conversion. The associated parameter in the value list must be the address of the fi rst location in an area of memory set aside to store the incoming string. Of course, a fi eld width may be specifi ed and this is the maximum number of characters that are read in, but remember that any extra characters are left unconsumed in the input buffer. A simple use of scanf() with%s conversions is illustrated in the following program.

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
```

Using gets()

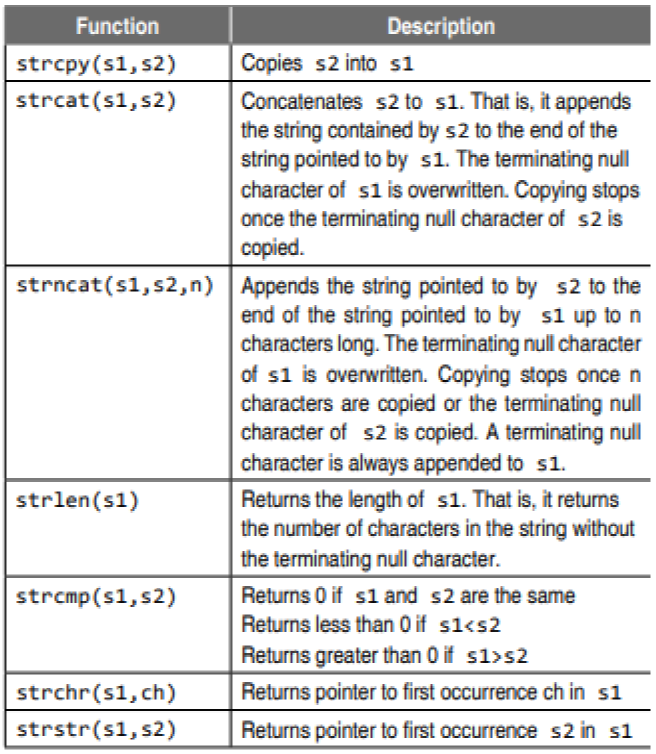
The best approach to string input is to use a library function called gets(). This takes the start address of an area of memory suitable to hold the input as a single parameter. The complete input line is read in and stored in the memory area as a null-terminated string. Its use is shown in the program below.

```
# include<stdio.h>
int main( ){
char name[ 25 ] ;
 printf ( "Enter your full name: " ) ;
 gets ( name ) ;
 puts ( "Hello!" ) ;
 puts ( name ) ;
 return 0 ;
}
```

The program and the output are self-explanatory except for the fact that, puts( ) can display only one string at a time (hence the use of two puts( ) in the program above). Also, on displaying a string, unlike printf( ), puts( ) places the cursor on the next line. Though gets( ) is capable of receiving only one string at a time, the plus point with gets( ) is that it can receive a multi-word string.

String Handling Functions:

The standard 'C' library provides various functions to manipulate the strings within a program. These functions are also called as string handlers. All these handlers are present inside <string.h> header file.



Function	Description
strcpy(s1,s2)	Copies s2 into s1
strcat(s1,s2)	Concatenates s2 to s1. That is, it appends the string contained by s2 to the end of the string pointed to by s1. The terminating null character of s1 is overwritten. Copying stops once the terminating null character of s2 is copied.
strncat(s1,s2,n)	Appends the string pointed to by s2 to the end of the string pointed to by s1 up to n characters long. The terminating null character of s1 is overwritten. Copying stops once n characters are copied or the terminating null character of s2 is copied. A terminating null character is always appended to s1.
strlen(s1)	Returns the length of s1. That is, it returns the number of characters in the string without the terminating null character.
strcmp(s1,s2)	Returns 0 if s1 and s2 are the same Returns less than 0 if s1<s2 Returns greater than 0 if s1>s2
strchr(s1,ch)	Returns pointer to first occurrence ch in s1
strstr(s1,s2)	Returns pointer to first occurrence s2 in s1

From the list given in Figure 15.2, we shall discuss functions strlen( ), strcpy( ), strcat( ) and strcmp( ), since these are the most commonly used. This will also illustrate how the library functions in general handle strings. Let us study these functions one-by-one.

strlen( )

This function counts the number of characters present in a string. Its usage is illustrated in the following program:

```
#include <stdio.h>
#include <string.h>
int main()
{
   char a[20]="Program";
   char b[20]={'P','r','o','g','r','a','m','\0'};

   // using the %zu format specifier to print size_t
   printf("Length of string a = %zu \n",strlen(a));
   printf("Length of string b = %zu \n",strlen(b));

   return 0;
}
```

Note that the strlen() function doesn't count the null character \0 while calculating the length.

strcpy( )

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of strcpy( ) in action...

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

int main() {
  char str1[20] = "C programming";
  char str2[20];

  // copying str1 to str2
  strcpy(str2, str1);

  puts(str2); // C programming

  return 0;
}
```

Note: When you use strcpy(), the size of the destination string should be large enough to store the copied string. Otherwise, it may result in undefined behavior.

strcat( )

This function concatenates the source string at the end of the target string. For example, “Bombay” and “Nagpur” on concatenation would result into a string “BombayNagpur”. Here is an example of strcat( ) at work.

```
#include <stdio.h>
#include <string.h>
int main() {
  char str1[100] = "This is ", str2[] = "programiz.com";

  // concatenates str1 and str2
  // the resultant string is stored in str1.
  strcat(str1, str2);

  puts(str1);
  puts(str2);

  return 0;
}
```

strcmp( )

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character-bycharacter until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, strcmp( ) returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pair of characters. Here is a program which puts strcmp( ) in action.

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

int main() {
 char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
 int result;

 // comparing strings str1 and str2
 result = strcmp(str1, str2);
 printf("strcmp(str1, str2) = %d\n", result);

 // comparing strings str1 and str3
 result = strcmp(str1, str3);
 printf("strcmp(str1, str3) = %d\n", result);

 return 0;
}
```

Output:

```
strcmp(str1, str2) = 1
strcmp(str1, str3) = 0
```

In the program,
strings str1 and str2 are not equal. Hence, the result is a non-zero integer.
strings str1 and str3 are equal. Hence, the result is 0.

Strupr()

The **strupr( )** function is used to converts a given string to uppercase.

Syntax: char *strupr(char *str);

Example:

```
// example of strupr() function.
#include<stdio.h>
#include<string.h>

int main()
{
        char str[ ] = "Hello FYCS class";
        //converting the given string into uppercase.
        printf("%s\n", strupr (str));
```

```
    return 0;
}
```

strlwr()

The **strlwr( )** function is a built-in function in C and is used to convert a given string into lowercase.

Syntax: **char *strlwr(char *str);**

Example:

```
// C program to demonstrate

// example of strlwr() function


#include<stdio.h>

#include<string.h>


int main()

{

        char str[ ] = "THIS IS MY CLASS";


        // converting the given string into lowercase.

        printf("%s\n",strlwr (str));


        return 0;

}
```

Functions

What is a Function?

A function is a self-contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions. As we noted earlier, using a function is something like hiring a person to do a specific job for you. Sometimes the interaction with this person is very simple; sometimes it's complex. Suppose you have a task that is always performed exactly in the same way—say a bimonthly servicing of your motorbike. When you want it to be done, you go to the service station and say, "It's time, do it now". You don't need to give instructions, because the mechanic knows his job. You don't need to be told how the job is done. You assume the bike would be serviced in the usual way, the mechanic does it and that's that. Let us now look at a simple C function that operates in much the same way as the mechanic. Actually, we will be looking at two things—a function that calls or activates the function and the function itself.

Imagine a program wherein a set of operations has to be repeated often, though not continuously, n times or so. If they had to be repeated continuously, loops could be used. Instead of inserting the program statements for these operations at so many places, write a separate program segment and compile it separately. As many times as it is needed, keep 'calling' the segment to get the result. The separate program segment is called a function and the program that calls it is called the 'main program'. C went one step further; it divided the entire concept of programming to a combination of functions. C has no procedures, only functions. scanf(), printf(), main(), etc. that have been used in programs so far, are all functions. C provides a lot of library functions; in addition, the programmers can write their own functions and use them. The special function called main() is where program execution begins. When a function is called upon, with or without handing over of some input data, it returns information to the main program or calling function from where it was called.

Why are Functions Needed?

The use of functions provides several benefits.

- First, it makes programs significantly easier to understand and maintain by breaking up a program into easily manageable chunks. Even without software engineering, functions allow the structure of the program to reflect the structure of its application.
- Secondly, the main program can consist of a series of function calls rather than countless lines of code. It can be executed as many times as necessary from different points in the main program. Without the ability to package a block of code into a function, programs would end up being much larger, since one would typically need to replicate the same code at various points in them.
- The third benefit is that well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions. This enables code sharing.
- Fourthly, functions can be used to protect data. This is related with the concept of local data. Local data is the data described within a function. They are available only within a function when the function is being executed.
- The fifth benefit of using functions is that different programmers working on one large project can divide the workload by writing different functions.

USING FUNCTIONS

Referring back to the Introduction, all C programs contain at least one function, called main() where execution starts. Returning from this function the program execution terminates and the returned value is treated as an indication of success or failure of program execution. When a function is called, the code contained in that function is executed, and when the function has fi nished executing, control returns to the point at which that function was called. The program steps through the statements in sequence in the normal way until it comes across a call to a particular function. At that point, execution moves to the start of that function—that is, the fi rst statement in the body of the function. Execution of the program continues through the function statements until it hits a return statement or reaches the closing brace marking the end of the function body. This signals that execution should go back to the point immediately after where the function was originally called. Functions are used by calling them from other functions. When a function is used, it is referred to as the 'called function'. Such functions often use data that is passed to them from the calling function. Parameters provide the means by which you pass information from the calling function into the called function. Only after the called function successfully receives the data can the data be manipulated to produce a useful result.

There are two types of functions in C i.e.

1. Library functions
2. User-defined Functions

Here, we will be talking about user-defined functions.

Function Prototype Declaration: All the header fi les contain declarations for a range of functions, as well as defi nitions for various constants. In a C program, a user-written function should normally be declared prior to its use to allow the compiler to perform type checking on the arguments used in its call statement or calling construct. The general form of this function declaration statement is as follows:

return_data_type function_name (data_type variable1, ...);

OR

return_data_type function_name (data_type_list);

There are three basic parts in this declaration.

- function_name: This is the name given to the function and it follows the same naming rules as that for any valid variable in C.
- return_data_type: This specifies the type of data given back to the calling construct by the function after it executes its specific task.
- data_type_list: This list specifies the data type of each of the variables, the values of which are expected to be transmitted by the calling construct to the function.

Function Definition:

The collection of program statements in C that describes the specifi c task done by the function is called a function defi nition. It consists of the function header and a function body, which is a block of code enclosed in parentheses. The defi nition creates the actual function in memory. The general form of the function definition is as follows

return_data_type function name(data_type variable1, data_type variable2,……)

{

/* Function Body */

```
}
```

The function header in this definition is return_data_type function name(data_type variable1, data_type variable2,……) and the portion of program code within the braces is the function body. Notice that the function header is similar to the function declaration but does not require the semicolon at the end. The list of variables in the function header is also referred to as the formal parameters. One point to be noted here is that the names do not need to be the same in the prototype declaration and the function definition. If the types are not the same then the compiler will generate an error. The compiler checks the types in the prototype statements with the types in the call to ensure that they are the same or at least compatible. A value of the indicated data type is returned to the calling function when the function is executed. The return data type can be of any legal type. If the function does not return a value, the return type is specified by the keyword void. The keyword void is also used to indicate the absence of parameters. So a function that has no parameters and does not return a value would have the following header.

```
void function_name(void)
```

A function with a return type specified as void should not be used in an expression in the calling function. Since it does not return a value, it cannot sensibly be part of an expression. Therefore, using it in this way will cause the compiler to generate an error message. There is no standard guideline about the number of parameters that a function can have. Every ANSI C compliant compiler is required to support at least 31 parameters in a function. However, it is considered bad programming style if a function contains an inordinately high (eight or more) number of parameters. The number of parameters a function has also directly affects the speed at which it is called—the more parameters, the slower the function call. Therefore, if possible, one should minimize the number of parameters to be used in a function.

Example:

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

  /* local variable declaration */
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

  /* local variable definition */
  int a = 100;
  int b = 200;
  int ret;

  /* calling a function to get max value */
  ret = max(a, b);

  printf( "Max value is : %d\n", ret );

  return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

  /* local variable declaration */
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

CALL BY VALUE MECHANISM

The technique used to pass data to a function is known as parameter passing. Data are passed to a function using one of the two techniques: pass by value or call by value and pass by reference or call by reference. In call by value, a copy of the data is made and the copy is sent to the function.

The copies of the value held by the arguments are passed by the function call. Since only copies of values held in the arguments are passed by the function call to the formal parameters of the called function, the value in the arguments remains unchanged. In other words, as only copies of the values held in the arguments are sent to the formal parameters, the function cannot directly modify the arguments passed. This can be demonstrated by deliberately trying to do so in the following example.

```
/* function definition to swap the values */
void swap(int x, int y) {

  int temp;

  temp = x; /* save the value of x */
  x = y;    /* put y into x */
  y = temp; /* put temp into y */

  return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example –

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main () {

  /* local variable definition */
  int a = 100;
  int b = 200;

  printf("Before swap, value of a : %d\n", a );
  printf("Before swap, value of b : %d\n", b );

  /* calling a function to swap the values */
  swap(a, b);

  printf("After swap, value of a : %d\n", a );
  printf("After swap, value of b : %d\n", b );

  return 0;
}
void swap(int x, int y) {

  int temp;
```

```
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

CALL BY REFERENCE MECHANISM

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

Let us now call the function **swap()** by passing values by reference as in the following example –

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
void swap(int *x, int *y) {

   int temp;

   temp = *x; /* save the value of x */
   *x = *y;    /* put y into x */
   *y = temp; /* put temp into y */

   return;
}
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

Concept of Global and Local Variables:

There are two common terms related to the visibility or accessibility of a variable. They are global and local variables. Actually global and local are the terms related with lifetime. Lifetime is the period during execution of a program in which a variable or function exists. It will be discussed in details later on in this section. Variables declared within the function body are called local variables. They have local scope. Local variables are automatically created at the point of their declaration within the function body and are usable inside the function body. These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. The existence of the local variables ends when the function completes its specific task and returns to the calling point. They are recreated each time a function is executed or called.

Variables declared outside of all the functions of a program and accessible by any of these functions are called global variables. The existence and region of usage of these variables are not confined to any specific function body. They are implemented by associating memory locations with variable names. Global variables are created at the beginning of program execution and remain in existence all through the period of the execution of the program. These variables are known to all functions in the program and can be used by these functions as many times as may be required. They do not get recreated if the function is recalled. Global variables do not cease to exist when control is transferred from a function. Their value is retained and is available to any other function that accesses them.

All global variables are declared outside of all the functions. There is no general rule for where outside the functions these should be declared, but declaring them on top of the code is normally recommended for reasons of scope, as explained through the given examples. If a variable of the same name is declared both within a function and outside of it, the function will use the variable that is declared within it and ignore the global one. If not initialized, a global variable is initialized to zero by default. As a matter of style, it is best to avoid variable names that conceal names in an outer scope; the potential for confusion and error is too great. Moreover, the use of global variables should be as few as possible. Consider the example below

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n", g);

  return 0;
}
```

Example2:

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n", a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n", c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b) {
  printf ("value of a in sum() = %d\n", a);
  printf ("value of b in sum() = %d\n", b);

  return a + b;
}
```

Recursion:

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

The formal definition is given below:
"A recursive function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call".

Recursion is like a top–down approach to problem solving; it divides the problem into pieces or selects one key step, postponing the rest. On the other hand, iteration is more of a bottom–up approach; it begins with what is known and from this construct the solution step by step.

What is needed for implementing recursion?

- Decomposition into smaller problems of same type
- Recursive calls must diminish problem size
- Necessity of base case
- Base case must be reached
- It acts as a terminating condition. Without an explicitly defined base case, a recursive function would call itself indefinitely.
- It is the building block to the complete solution. In a sense, a recursive function determines its solution from the base case(s) it reaches.

The recursive algorithms will generally consist of an if statement with the following form:

if(this is a base case)

then solve it directly

else redefine the problem using recursion.

Example:

```
if (test_for_base)
{
   return some_value;
}
else if (test_for_another_base)
{
   return some_another_value;
}
else
{
   // Statements;
   recursive call;
}
```

Example2: Fibonacci series

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
   int n,f;
   printf("Enter the value of n?");
   scanf("%d",&n);
   f = fibonacci(n);
   printf("%d",f);
}
int fibonacci (int n)
{
   if (n==0)
   {
   return 0;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

Example3: Number Factorial

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```