Functions

Subtopics

- **Functions**
- **Function Declaration**
- **Function Arguments**
- **Return Statements and values**

Function

- Functions are building blocks of the programs.
- They make the programs more modular and easy to read and manage.
- Functions can be called several times in the same program, allowing the code to be reused.
- All C programs must contain the function main( ) that too only once.
- The execution of the program starts from the function main( ).

Advantages of Using Functions

- Easier to understand , debug and test.
- It facilitates top-down modular programming.
- The length of source program can be reduced by using function at appropriate place.
- It is easy to locate and isolate a faulty function for further investigation.
- A function may be used by other programs.

Functions

C functions can be classified into two categories

- Library Functions((e.g., pow, sqrt etc.) are usually grouped into specialized libraries (e.g. stdlib, math, etc.)
- User Defined Functions

Difference:

Library functions are inbuilt, where as user defined function has to be developed by the user at the time of writing a program.

Need of Functions

- Functions are used because of following reasons
  a) To improve the readability of code.
  b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
  c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.
  d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Elements for the user defined Functions

- Function definition
- Function Call
- Function declaration/Prototype

Form of a function

Syntax: Function Declaration

```
return_type function_name(parameter list) ;
```

Syntax : Function Call

```
int main()
{
Function_name(parameter list);
}
```

Syntax: Function Definition

```
Function_type function_name(parameter list)
{
 local variable declaration;
 executable statement1;
  executable statement 2;
  return;
   }
```

The General Form of a Function is

returntype function-name(parameter-list)
{
Body of the function
}

- The function can return any type of data like int, char, float, double etc.
- The parameter list is comma separated list of variable names and their associated types the receive values of the arguments when the function is called.
- A function may be without parameters or with parameters.

fun(type varname1, type varname2…….,type var name N)
Example:
int fun1(int i,int j,int k) is the function whose name is fun1, and it takes three integer i,j,k and returns int .

Function prototyping

- The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

- Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself.

Function prototyping

- In function declaration , the names of the arguments are dummy variables and then they are optional

  Ex:
  int fun1(int, int, int);
  acceptable to the place of declaration, here compiler only checks the type of arguments when the function is called.

- We can also declare a function with an empty argument list
  void display();
  i.e. the function does not pass any parameters
  Similar to void display(void)

FUNCTION ARGUMENTS

Formal arguments are the parameters present in a function definition which may also be called as dummy arguments or parametric variables. When the function is invoked, the formal parameters are replaced by the actual parameters.

- They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- **Actual arguments**: An actual argument is a variable or an expression contained in a function call that replaces the formal parameter which is a part of the function declaration.

- Variables that are defined within a function are called local variables.

```
int fun1(int i,int j, int k)
{
int sum;
sum = i+j+k;
return sum;
}
```

Here sum is the local variable.

Example

```
#include <stdio.h>
int addition(int, int);    //Function Prototype
int main()
{
    int var1, var2;
    printf("Enter number 1: ");
    scanf("%d%d",&var1,&var2);
    int res = addition(var1, var2);//Function Calling
    printf ("Output: %d", res);
    return 0;
}
int addition(int num1, int num2)//Function Definition
{
    int sum;     /* Arguments are used here*/
    sum = num1+num2;
    return sum;
}
```

GCD of two numbers

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
  int num1, num2;
  printf("Enter two numbers : ");
  scanf("%d %d",&num1, &num2);
  int result = gcd(num1, num2);
  printf("GCD of %d and %d = %d", num1, num2, result);
  return 0;
}
```

```
int gcd(int a, int b)
{
  int hcf,i;
  for( i=1; i<=a && i<=b; i++)
  {
    if(a%i==0 && b%i==0)
    {
      hcf = i;
    }
  }
  return hcf;
}
```

Function Call

- Parameters Pass/Call by Value
- Parameters Pass/Call by Reference

Call by value and Call by Reference

- Copies of the arguments are created .
- The parameters are mapped to the copies of the arguments created.
- The changes made to the parameter do not affect the arguments.
- **Call by Reference**: Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller.

Example

```
#include <stdio.h>
void call_by_value(int x)
{
printf("Inside call_by_value x = %d before adding 10.\n", x);
x += 10;
printf("Inside call_by_value x = %d after adding 10.\n", x);
}

int main() {
int a=10;
printf("a = %d before function call_by_value.\n", a);
call_by_value(a);
printf("a = %d after function call_by_value.\n", a);
return 0;
}
```

Call by reference example

```
void swapx(int*, int*);

// Main function
int main()
{
    int a = 10, b = 20;

    // Pass reference
    swapx(&a, &b);

    printf("a=%d b=%d\n", a, b);

    return 0;
}
```

```
// Function to swap two variables
// by references
void swapx(int* x, int* y)
{
    int t;

    t = *x;
    *x = *y;
    *y = t;

    printf("x=%d y=%d\n", *x, *y);
}
```

What is recursion?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type

When you turn this into a program, you end up with functions that call themselves (*recursive functions*)

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

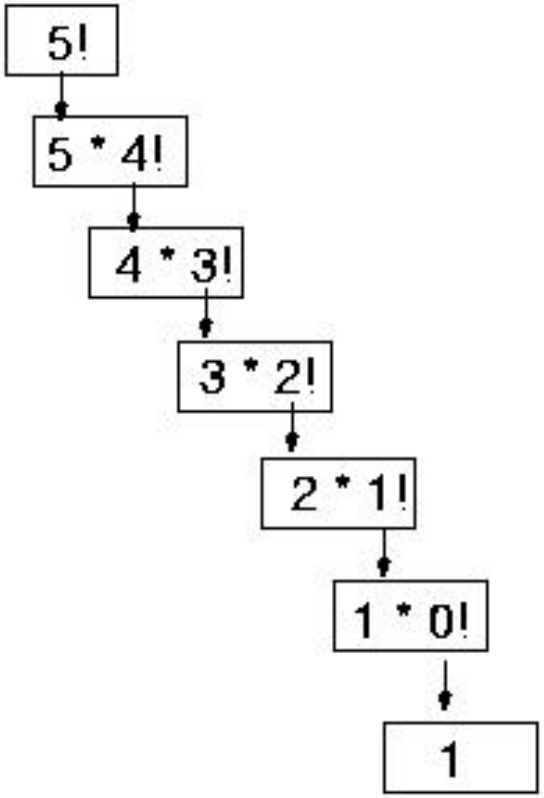
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases}$$

(*recursive* solution)

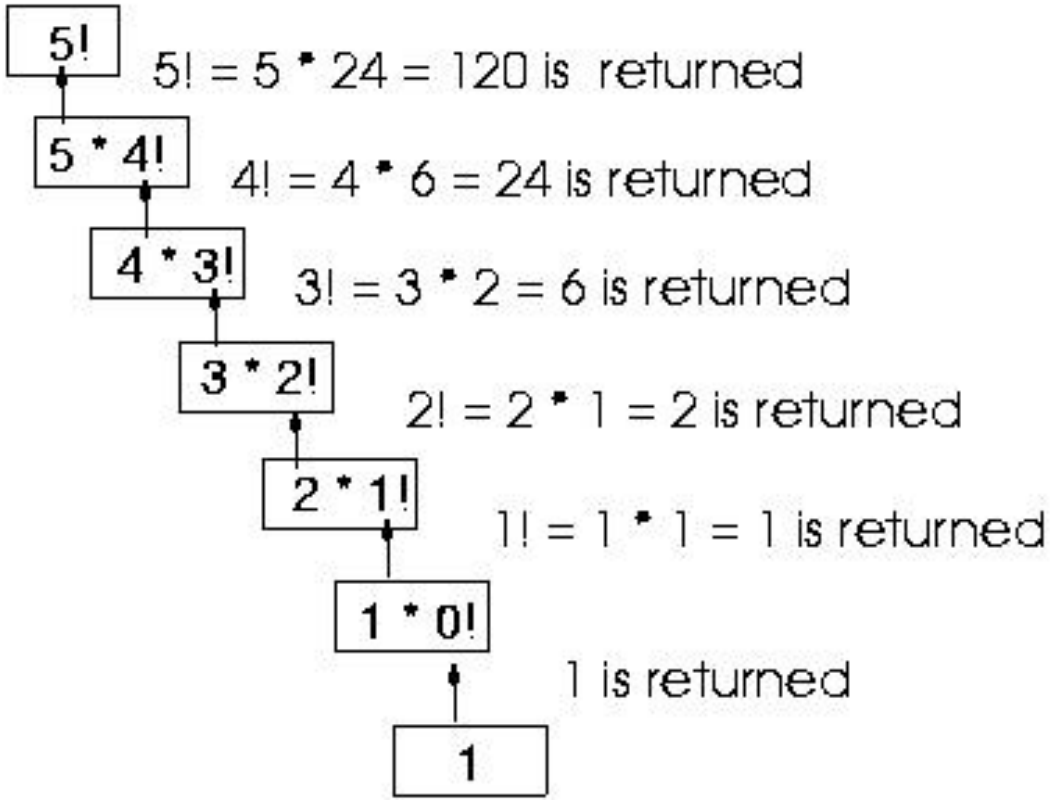
Coding the factorial function

- Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```



Final value = 120



Coding the factorial function (cont.)

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Coding the factorial function (cont.)

- Recursive Implementation

```
#include<stdio.h>

long int multiplyNumbers(int n);
int main() {
   int n;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   printf("Factorial of %d = %ld", n, multiplyNumbers(n));
   return 0;
}

 long int multiplyNumbers(int n) {
   if (n>=1)
      return n*multiplyNumbers(n-1);
   else
      return 1;
}
```