COMPUTER PROGRAMMING - I

Functions

- C provides ways to divide a long and continuous program into small modules that are related in some manner.
- These modules are called functions

Functions

```
main( )
{
printf ( "\n We are in main" ) ;
 message( ) ;
 printf ( "\nWe are back in main" ) ;
}
message( )
{
 printf ( "\n Hello from message function " ) ;
}
```

```
main( )
{
 printf ( "\nI am in main" ) ;
 italy( ) ;
 brazil( ) ;
 argentina( ) ;
}
```

```
italy( )
{
 printf ( "\nI am in italy" ) ;
}
brazil( )
{
 printf ( "\nI am in brazil" ) ;
}
argentina( )
{
 printf ( "\nI am in argentina" ) ;
}
```

Functions

- Any C program contains at least one function.
- If a program contains only one function, it must be main( ).
- If a C program contains more than one function, then one (and
- only one) of these functions must be main( ), because program
- execution always begins with main( ).
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in main( ).
- After each function has done its thing, control returns to main( ). When main( )runs out of function calls, the program ends.

Why use functions

- Avoid rewriting of same code again and again
- Improvement of overall organization of a program
- Facilitation of team work
- Simplification of task like debugging and testing

Flow of control in Functions

```
main( )
{
 printf ( "\n I am in main" ) ;
 italy( ) ;
 printf ( "\n I am finally back in main" ) ;
}
italy( )
{
 printf ( "\nI am in italy" ) ;
 brazil( ) ;
 printf ( "\nI am back in italy" ) ;
}
```

Flow of control in Functions

- brazil( )
- {
- printf ("\n I am in brazil") ;
- argentina( ) ;
- }
- argentina( )
- {
- printf ("\nI am in argentina") ;
- }

Flow of control in Functions

- I am in main
- I am in italy
- I am in brazil
- I am in argentina
- I am back in italy
- I am finally back in main

Types of Functions

- There are basically two types of functions:
- **Library functions/Built in functions:**
- These are available as part of C library.
- Ex. printf( ), scanf( )etc.

- **User-defined functions:**
- These are written by a programmer to solve a problem
- Ex. argentina( ), brazil( )etc

General form

- Return-type function-name(argument_list)
- {
- local variable;
- statements;
- return(expression);
- }
- double funct(int a, int b, double c)

General form

- data- type function- name(type1 a1,type2 a2,…..typeN aN)
- ------function header----
- {
- ----------(body of the function)
- --------
- }

- Function header + function body = function Definition

Functions

- A function which is defined cannot be executed by itself.
- It has to be invoked by another function.
- It can be main() or any other function
- A function which invokes another function is called calling function or caller function and the function which is invoked is termed as Called function

Category of functions

- A function may belong to one of the following categories.
- 1) Functions with no arguments and no return values.
- 2) Functions with arguments and no return values.
- 3) Functions with arguments and return values.
- 4) Functions with no arguments and return values.

Functions with no arguments and no return values.

- A function does not receive any data from the calling function.
- Similarly, it does not return any value.
- If the function is not returning any value, its return type will be void.

Functions with no arguments and no return values.

```
#include<stdio.h>
void add();
int main()
{
   add();
   return 0;
}
```

Functions with no arguments and no return values.

```
void add()
{
   int a,b,sum;
  printf("\n Enter any two positive integer");
  scanf("%d%d",&a,&b);
  sum = a+b;
  printf("\nSum =%d",sum);

}
```

Functions with arguments and no return values.

```
#include<stdio.h>
void add(int,int);
int main()
{
  int a,b;
  printf("\n Enter any two positive integer");
  scanf("%d%d",&a,&b);
   add(a,b);
   return 0;
}

void add(int x, int y)
{
   int sum;
    sum = x+y;
  printf("\nSum =%d",sum);

}
```

Functions with arguments and no return values.

```
#include<stdio.h>
void add(int,int);
int main()
{
  int a,b;
  printf("\n Enter any two positive integer");
  scanf("%d%d",&a,&b);
   add(a,b);
   return 0;
}
```

Functions with arguments and no return values.

```
void add(int x, int y)
{
   int sum;
    sum = x+y;
  printf("\nSum =%d",sum);

}
```

Function Parameters

- The parameters passed by the caller function are called as actual parameters
- The parameters received by the called function are called as formal parameters
- In above example two variable a, b are passed to function during function call (here a and b are called actual parameters)
- And values of these arguments are accepted by arguments x and y in function definition (here x and y are called formal parameters)

Functions Parameters

- **Rules of passing parameters**
- The number of actual and formal parameters must always be same
- The data types of actual and formal parameters must always be same

- If a function is called before its definition, then a prototype declaration of that function must be written before calling the function

Functions with arguments and return values.

```
#include<stdio.h>
int add(int,int);
int main()
{
  int a,b,sum;
  printf("\n Enter any two positive integer");
  scanf("%d%d",&a,&b);
   sum = add(a,b);
   printf("\nSum =%d",sum);
  return 0;
}
```

Functions with arguments and return values.

```
int add(int x, int y)
{
   int z;
   z = x+y;
   return z;
}
```

Functions with no arguments and return values.

```
#include<stdio.h>
int add( );
int main()
{
  int sum;
   sum = add();
   printf("\nSum =%d",sum);
  return 0;
}
```

Functions with no arguments and return values.

```
int add(int x, int y)
{
   int a,b,sum;
  printf("\n Enter any two positive integer");
  scanf("%d%d",&a,&b);
  sum = a+b;
   return sum;
}
```

Quiz time

```
main( )
{
 int a = 30 ;
 fun ( a ) ;
 printf ( "\n%d", a ) ;
}
fun ( int b )
{
 b = 60 ;
 printf ( "\n%d", b ) ;
}
```

Quiz time

```
main( )
{
 int i = 20 ;
 display ( i ) ;
}
display ( int j )
{
 int k = 35 ;
 printf ( "\n%d", j ) ;
 printf ( "\n%d", k ) ;
}
```

Quiz time

```
int main( )
{
 int i = 20 ;
 display ( i ) ;
 printf ( "\n%d", i ) ;
}
void display ( int i )
{
  i = 30 ;
 printf ( "\n%d", i ) ;
}
```

Practice

```
// Find factorial using functions
int fact(int n)
{
    int i,prod =1;
    for(i=1;i<=n;i++)
    {
        prod = prod*i;
    }
    return prod;
}
```

Practice

```
int main()
{
    int n,factorial;
    printf("Enter the number");
    scanf("%d",&n);
    factorial = fact(n);
    printf("Factorial of a number %d = %d",n,factorial);
}
```

Practice

```
//Print Fibonacci series using functions
//0 1 1 2 3 5 8 13 21 34 55
int fibo(int n)
{
   int i,a = 0, b=1, sum=0;

 printf("\t %d",a);
   printf("\t %d",b);

 for(i=2;i<n;i++)
   {

       sum= a+b;
        printf("\t %d",sum);
        a=b;
       b=sum;

   }
 return sum;
}
```

Practice

```
int main()
{
    int n,fibonacci;
    printf("Enter the number");
    scanf("%d",&n);
    fibonacci = fibo(n);
    printf("\n fibonacci of a number %d = %d",n,fibonacci);
}
```

- Passing arrays to functions

Binary Search Algorithm

- May only be used on a sorted array.
- Eliminates one half of the elements after each comparison.
- Locate the middle of the array
- Compare the value at that location with the search key.
- If they are equal - done!
- Otherwise, decide which half of the array contains the search key.
- Repeat the search on that half of the array and ignore the other half.
- The search continues until the key is matched or no elements remain to be searched.

Binary search Algorithm

```
int Binarysearch(int low, int high, int key)
while(low <= high)
        {
                mid = (low + high)/2;
                if(array[mid] < key)
                {
                        low = mid + 1;
                }
                else if(array[mid] == key)
                {
                        return mid;
                }
                else if(array[mid] > key)
                {
                        high = mid-1;
                }

        }
        return -1;
```

Binary Search

- int A[ ] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; Key = 9
- **The algorithm will proceed in the following manner**.
- **Step1:** low = 0, high= 10, mid = (0 + 10)/2 = 5
- Now, Key= 9 and A[mid] = A[5] = 5
- **Step 2**: A[5] is less than Key, therefore, we will now search for the value in the later half of the array.
- Now, low= mid+ 1 = 6, high= 10, mid= (6 + 10)/2 =16/2 = 8
- Now, Key= 9 and A[mid] = A[8] = 8
- **Step3:** A[8] is less than key, therefore, we will now search for the value in the later half of the array.
- Now, low= mid+ 1 = 9, high= 10, mid= (9 + 10)/2 = 9
- Now Key= 9 and A[mid] = 9.
- Therefore, Value is found

Practice Problems

Program to find Standard Deviation by Using Array

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

1. Calculate Mean (the simple average of the numbers)
2. Then for each number: subtract the Mean and square the result
3. Then calculate the mean of **those** squared differences.
4. Take the square root of that and we are done!

```
// Finding mean and standard deviation
int main()
{
    int n, i;
    float data[100];
    float sum=0.0, sum_deviation=0.0,mean;
    printf("Enter number of elements( should be less than 100): ");
    scanf("%d",&n);
    printf("Enter elements: ");
    for(i=0; i<n; ++i)
        scanf("%f",&data[i]);
    printf("\n");
```

```
    for(i=0; i<n;i++)
    {
        sum= sum + data[i];
    }
    mean=sum/n;
    for(i=0; i<n;i++)
    sum_deviation+=(data[i]-mean)*(data[i]-mean);
    sum_deviation =sqrt(sum_deviation/n);

    printf("Standard Deviation = %f",sum_deviation);
    return 0;
}
```

```
Enter number of elements( should be less than 100): 8
Enter elements: 65
9
27
78
12
20
33
49
Standard Deviation = 23.510303
```

Passing Arrays into functions

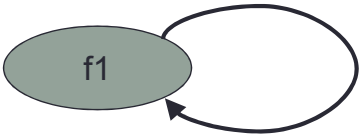
```
// Finding Mean and standard deviation by passing arrays into functions
float CalculateStDev(float data[],int n)
{
    float mean=0.0, sum_deviation=0.0;
    int i;
    for(i=0; i<n;++i)
    {
        mean+=data[i];
    }
    mean=mean/n;
    for(i=0; i<n;++i)
    sum_deviation+=(data[i]-mean)*(data[i]-mean);
    return sqrt(sum_deviation/n);
}
```

Passing Arrays into functions

```
int main()
{
    int n, i;
    float data[100];
   float deviation;
    printf("Enter number of elements( should be less than 100): ");
    scanf("%d",&n);
    printf("Enter elements: ");
    for(i=0; i<n; ++i)
        scanf("%f",&data[i]);
    printf("\n");
    deviation = CalculateStDev(data,n);
     printf("Standard deviation = %f",deviation);
    return 0;
}
```

Recursion

- *Recursion*: defining something in terms of itself



- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.

Basic Recursions

- Break a problem into smaller identical problems
  - Each recursive call solves an identical but smaller problem.
- Stop the break-down process at a special case whose solution is obvious, (termed **a base case**)
  - Each recursive call tests the base case to eventually stop.
  - Otherwise, we fall into an infinite recursion.

Content of a Recursive Method

- **Base case(s).**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.
- ***Recursive calls.***
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Factorial

- ***Non-recursive* definition:** *n factorial*(*n!*) is the product of the integers between 1 and n inclusive when n >=1;
- 0! = 1

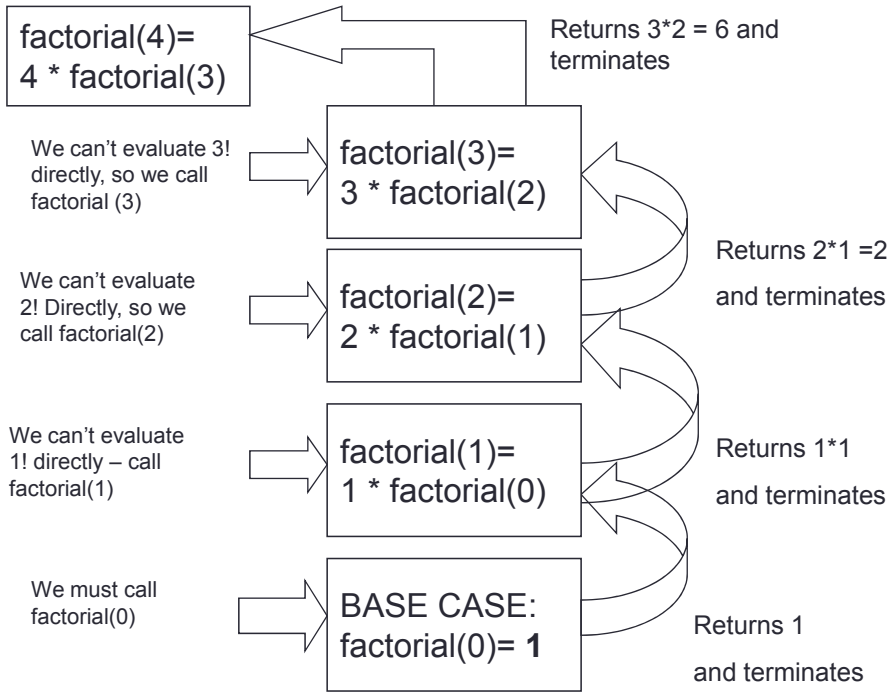
```
int factorial (int n)
{
  int prod =1;  // 1!
  for(int  i = 1; i<=n; i++)
          prod = prod * i;
  return  prod;
}
```

- ***Recursive* definition for computing n!:**
- •0! = 1 *(base case)*
- •n! = n * **(n-1)!** for all n >0 (general case)

```
int factorial (int n)
{
  //base case
  if (n ==0) return 1;

  //recurrence case
  return n * factorial (n – 1);
}
```

Trace of a call to Factorial: int z = factorial(4)



***finally*, factorial(4) computes 4*6, returns 24, and terminates**

Fibonacci

- Fibonacci series can be given as
- 0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

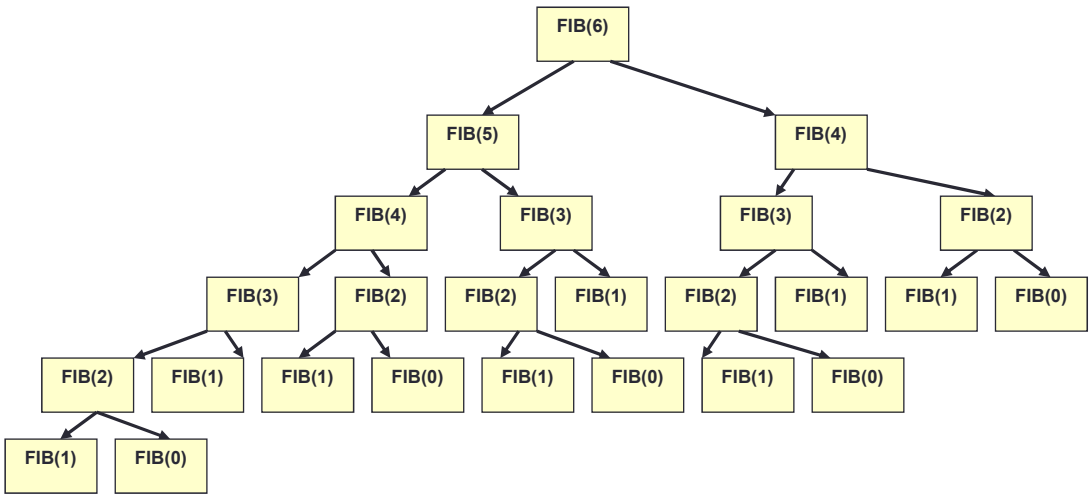
Fibonacci – Iterative version

```
void fib(int n)
 {
          if (n<=1)
            return n;
  int a = 0;
  int b= 1;
 printf("%d %d",a,b);
  for (int i = 2; i <= n; i++)
 {
   c  = a+ b;
    printf("%d",c);
     a = b;
     b=c
      }
  }
```

Fibonacci – Recursive method

- int fib(int n)
- {
- if (n<=1)return n;
- return fib(n-1) + fib(n-2);
- }
- In this instance, there are *two* base cases, and the general problem is solved in terms of *two* smaller versions of the problem

Fibonacci – Recursive method



Gcd

The greatest common divisor(GCD) of two numbers (integers) is the largest integer that divides both the numbers.
We can find GCD of two numbers recursively by using Euclid's algorithm

GCD(a,b) = b, if b divides a
GCD(b, a mod b), otherwise

(Here we assume that a>b. If a<b, then interchangea and b)

GCD – Non Recursive Method

```
int  gcd_iter(int u,  int v) {
  int  t;
  while (v)
 {
    t = u;
    u = v;
    v = t % v;
  }
  return u < 0 ? -u : u; /* abs(u) */
```

GCD –Recursive Method

Precondition: a and b both >0
Assume a>b

```
Int gcd(int a, int b)
{
        int ans;
        if(a % b == 0)
        ans = b;
        else
        ans = gcd(b, a%b)
        return (ans);
}
```