

CS101 Introduction to computing

Function

A. Sahu and P. Mitra

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- **Function**
- Calling, Definition
- Parameter Passing
- Local Variable
- Scope Rules

Functions

- Modularize a program
- All variables declared inside functions are local variables : Known only in function defined
- Parameters: Communicate info. between functions
- Function Benefits
 - Divide and conquer : Manageable program development
 - Software reusability : Use existing functions as building blocks for new programs and
 - Abstraction : hide internal details (library functions)
 - Avoids code repetition

Functions

- A C program is made up of one or more functions, one of which is `main()`.
- Execution always begins with `main()`
 - No matter where it is placed in the program.
- `main()` is located before all other functions.
- When program control encounters a function name, the function is **called (invoked)**.
 1. Program control passes to the function.
 2. The function is executed.
 3. Control is passed back to the calling function.

Sample Function Call

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

```
int main ( ) {
```

```
printf( "Hello World! \n" );
```

```
return 0 ;
```

```
}
```

printf is the name of a
predefined function in the
stdio library

this statement is
is known as a **function call**

this is a string we are **passing**
as an **argument (parameter)** to
the printf function

Functions (con't)

- We have used three predefined functions so far:
 - printf, scanf, pow, sqrt, abs, sin, cos
- Programmers can write their own functions.
- Typically, each module in a program's design hierarchy chart is implemented as a function.

Sample -Defined Function

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

```
void PrintMessage(void) ;
```

```
int main() {
```

```
    PrintMessage() ;
```

```
    return 0 ;
```

```
}
```

```
void PrintMessage(void) {
```

```
    printf("A MSG : \n\n") ;
```

```
    printf("Nice day! \n") ;
```

```
}
```

**Function
Prototype/
Declaration**

Function Call

**Function
Header**

**Function
Body or
Definition**

The Function Prototype

- Informs the compiler that there will be a function defined later that:

```
void PrintMessage(void) ;
```

Returns this
type

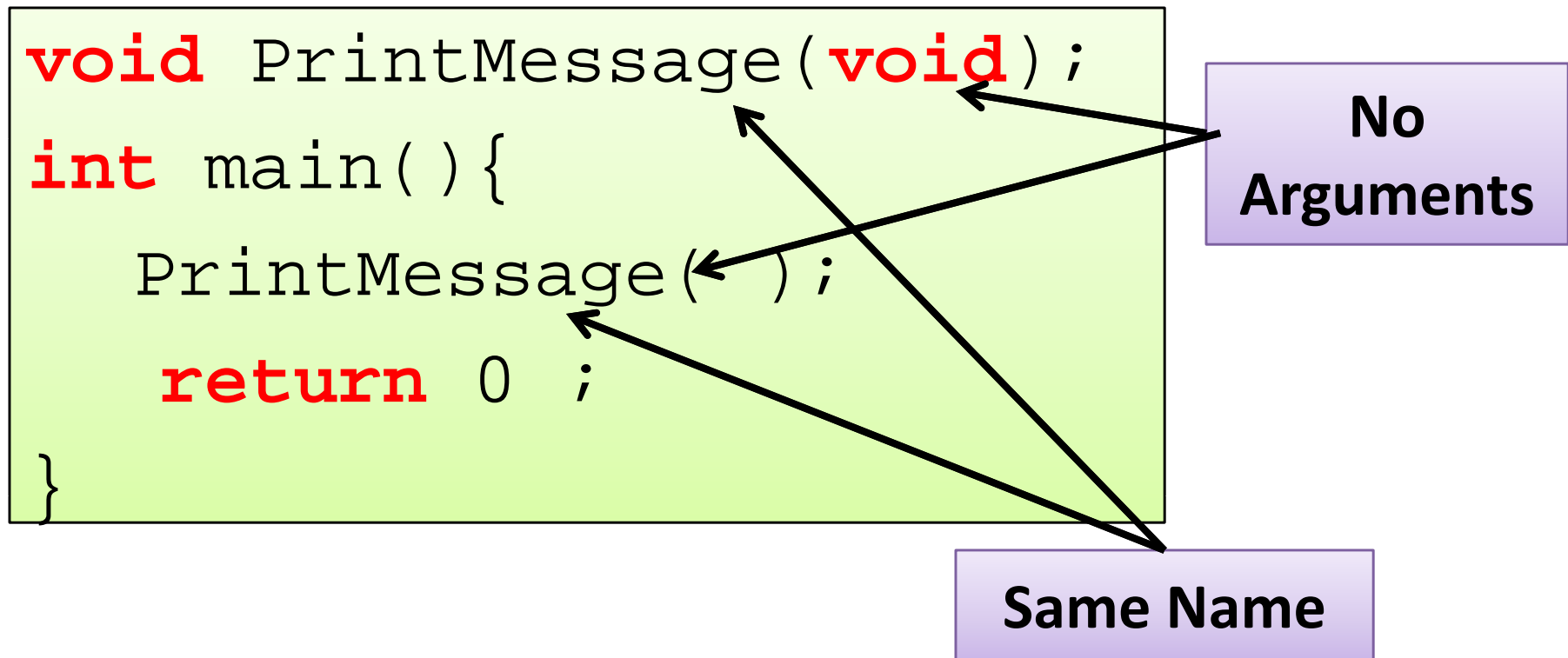
Has this name

Takes these type of
arguments in order

- Needed because the function call is made before the definition -- the compiler uses it to see if the call is made properly

The Function Call

- Passes program control to the function
- Must match the prototype in name, number of arguments, and types of arguments



The Function Definition

- Control is passed to the function by function call
 - The statements within the function body will then be executed

```
void PrintMessage(void) {  
    printf("A MSG : \n\n");  
    printf("Nice day! \n");  
}
```

- After statements in the function have completed
 - Control is passed back to the **calling function**
- In this case main()
 - Note that the calling function does not have to be main() .

General Function Definition Syntax

```
type functionName ( parameter1, . . . , parametern ) {  
    variable declaration(s)  
    statement(s)  
}
```

- If there are no parameters
 - either `functionName()` OR `functionName(void)`
- There may be no variable declarations.
- If the **function type** (**return type**) is void, a return statement is not required
 - Permitted: **return**; OR **return()**;

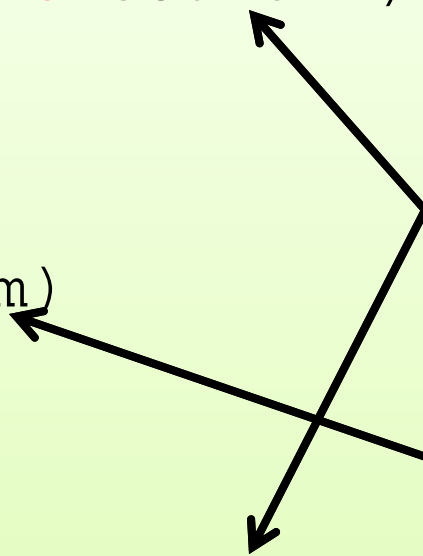
Input Parameters to Function

```
void PrintMessage(int counter) ;
```

```
int main ( ){  
    int 10;  
    PrintMessage(num)  
    return 0 ;  
}
```

```
void PrintMessage(int counter) {  
    int i ;  
    for (i=0;i<counter; i++)  
        printf ("Nice day!\n") ;  
}
```

matches the one
formal parameter
of type int



one argument
of type int

Functions Can Return Values : Example

```
#include <stdio.h>

float AverageTwo(int num1, int num2);

int main() {
    float ave ;
    int value1 = 5, value2 = 8 ;
    ave=AverageTwo(value1, value2) ;
    printf("The average of %d & %d
           is %f\n",value1,value2,ave);
    return 0 ;
}

float AverageTwo (int num1, int num2) {
    return (float)((num1+num2)/2.0) ;
}
```

Temp Convert Function in C

```
double CtoF ( double paramCel ) {  
    return paramCel*1.8+32.0;  
}
```

- This function takes an input parameter
 - Called paramCel (temp in degree Celsius)
- Returns a value
 - that corresponds to the temp in degree Fahrenheit

How to use a function?

```
#include <stdio.h>
double CtoF( double );
/* Purpose: to convert temperature
 * from Celsius to Fahrenheit *****/
int main() {
    double c, f;
    printf("Enter the degree (in Celsius): ");
    scanf("%lf", &c);
    f = CtoF(c);
    printf("Temperature (in Fahrenheit)
           is %lf\n", f);
}

double CtoF ( double paramCel) {
    return paramCel * 1.8 + 32.0;
}
```

Terminology

- Declaration

```
double CtoF( double ) ;
```

- Invocation (Call)

```
double CtoF( double ) ;
```

- Definition

```
double CtoF( double paramCel ) {  
    return paramCel*1.8 + 32.0 ;  
}
```


Modularity: Example

Declarations

```
#include <stdio.h>

double GetTemp();
double CelsToFahr(double);
void DispRes(double, double);

int main(){
    double TempC, TempF;

    TempC=GetTemp();
    TempF=CelsToFahr(TempC);
    DispRes(TempC, TempF);

    return 0;
}
```

Invocations

```
double CelsToFahr(double Tem){
    return (Tem * 1.8 + 32.0);
}
```

```
double GetTemp (){
    double Temp;
    printf("Please enter temp in
           degrees Celsius:");
    scanf("%lf", &Temp);
    return Temp;
}
```

```
void DispRes(double CTemp,
             double FTemp){
    printf("Original: %5.2f
           C\n", CTemp);
    printf("Equivalent: %5.2f
           F\n", FTemp);
}
```

Abstractions

- We are hiding details on *how* something is done in the function implementation
 - Put in library ☺ ☺ : do you require to know code for printf ? **No**

```
#include <stdio.h>

int main(){
    double TempC, TempF;

    TempC=GetTemp();
    TempF=CelsToFahr(TempC);
    DispRes(TempC,TempF);

    return 0;
}
```

```
double CelsToFahr(double Tem){
    return (Tem * 1.8 + 32.0);
}
```

```
double GetTemp (){
    double Temp;
    printf("Please enter temp in
           degrees Celsius:");
    scanf("%lf", &Temp);
    return Temp;
}
```

```
void DispRes(double CTemp,
             double Ftemp){
    printf("Original: %5.2f
           C\n", CTemp);
    printf("Equivalent: %5.2f
           F\n", FTemp);
}
```

Parameter Passing

- **Actual parameters** are the parameters that appear in the function call

```
ave =AverageTwo( value1 , value2 ) ;
```

- **Formal parameters** are the parameters that appear in the function header

```
float AverageTwo( int num1 , int num2 )
```


- Actual and formal parameters are matched by position.
- Each formal parameter receives the value of its corresponding actual parameter.

Parameter Passing (cont..)

- Corresponding actual and formal parameters
 - Do not have to have the same name, but they may.
 - Must be of the same data type, with some exceptions, Exception example

```
int fact(int N){ //Code for Fact;}
main(){
    float X=10.34;
    printf("fact of %f is %d",x, fact( X ));
}
```

auto conversion



Local Variables

- Functions only “see” (have access to) their own **local variables**. This includes `main()`
- Formal parameters are declarations of local variables.
 - The values passed are assigned to those variables.
- Other local variables can be declared within the function body.

Parameter Passing and Local Variables

```
int main() {  
    float ave ;  
    int v1=5, v2=8 ;  
    ave=AvgOfTwo(v1, v2);  
    printf ("The average  
           is %f\n", ave);  
    return 0 ;  
}
```

```
float AvgOfTwo(int n1,  
               int n2) {  
    float average;  
    average=(n1+n2)/2;  
    return average;  
}
```

Local copy of variables

5

v1

8

v2

6.5

ave

5

n1

8

n2

6.5

average

Same Name, Still Different Memory Locations

```
int main() {  
    float ave ;  
    int n1=5, n2=8 ;  
    ave=AvgOfTwo(n1, n2);  
    printf ("The average  
           is %f\n", ave);  
    return 0 ;  
}
```

```
float AvgOfTwo(int n1,  
               int n2) {  
    float average;  
    average=(n1+n2)/2;  
    return average;  
}
```

Local copy of variables

5

n1

8

n2

6.5

ave

5

n1

8

n2

6.5

average

Changes to Local Variables Do NOT Change Other Variables with the Same Name

```
int main() {  
    int n1=5;  
    AddOne(n1);  
    printf ("In main  
           n1 is %d\n",n1);  
    return 0 ;  
}
```

5

n1

```
void AddOne (int n1) {  
    n1=n1+1;  
    printf ("In AddOneF  
           n1 is %d\n",n1);  
    return;  
}
```

6

n1

Local copy of variables

OUTPUT

In AddOneF n1 = 6

In main n1 = 5



Solution : use Pass by reference

```
int main() {  
    int n1=5;  
    int *Pn1;  
    Pn1=&n1;  
    AddOne(Pn1);  
    printf ("In main  
           n1 is %d\n",n1);  
    return 0 ;  
}
```

5	&n1
n1	Pn1

```
void AddOne(  
    int *Pn1) {  
    *Pn1=*Pn1+1;  
    printf ("In AddOneF  
           n1 is %d\n", *Pn1);  
    return;  
}
```

&n1

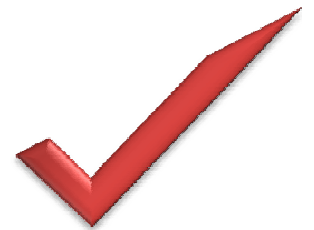
Pn1

Local copy of
Ptr variables

OUTPUT

In AddOneF n1 = 6

In main n1 = 6



Changes to Local Variables Do NOT Change Other Variables with the Same Name

```
int main() {  
    int n1=5, n2=10;  
    swap(n1,n2);  
    printf ("In main n1=  
        %d n2=%d\n",n1,n2);  
    return 0 ;  
}
```

5	10
n1	n2

```
void swap(int n1,  
          int n2) {  
    int tmp;  
    tmp=n1; n1=n2;n2=tmp;  
    printf ("In main n1=  
        %d n2=%d\n",n1,n2);  
}
```

Local copy of variables

10	5	5
n1	n2	tmp

OUTPUT

In swap n1 = 10 n2 =5

In main n1 = 5 n2 =10



Use Pass by Address/Reference

```
int main() {  
    int n1=5, n2=10;  
    swap(&n1,&n2);  
    printf ("In main n1=  
        %d n2=%d\n",n1,n2);  
    return 0 ;  
}
```

5	10
n1	n2

```
void swap(int *Pn1,  
          int *Pn2) {  
    int tmp;  
    tmp=*Pn1;  
    *Pn1=*Pn2;*Pn2=tmp;  
    printf ("In main n1=  
        %d n2=%d\n",n1,n2);  
}
```

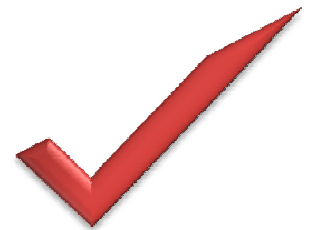
Local copy of variables

&n1	&n2	5
Pn1	Pn2	tmp

OUTPUT

In swap n1 = 10 n2 =5

In main n1 = 10 n2 =5



Passing Array to Function

`/(const float *age) (float *age) (float age[6]) same`



```
float average(float age[]){  
    int i; float avg, sum = 0.0;  
    for (i = 0; i < 6; ++i) {  
        sum = sum + age[i]; age[i]=1;  
    }  
    avg = (sum / 6); return avg;  
}
```

```
int main(){  
    float avg, age[]={23.4,55,22.6,3,40.5,18};  
    int i;  
    avg = average(age);  
    printf("Average age=%.2f\n", avg);  
    for(i=0;i<6;++i) printf("%1.2f",age[i]);  
    return 0 ;  
}
```

Storage Classes

- **Storage class specifiers : static, register, auto, extern**
 - Storage duration – how long an object exists in memory
 - Scope – where object can be referenced in program
 - Linkage – specifies the files in which an identifier is known
- **Automatic storage**
 - Object created and destroyed within its block
 - auto: default for local variables `auto double x, y;`
 - register: tries to put variable into high-speed registers
 - Can only be used for automatic variables

Automatic Storage

- Object created and destroyed within its block
- auto: **default for local variables**

```
auto double x, y; //same as double x, y
```

- **Conserving memory**
 - because automatic variables exist only when they are needed.
 - They are created when the function in which they are defined is entered
 - and they are destroyed when the function is exited
- **Principle of least privilege**
 - Allowing access to data only when it is absolutely needed.
 - Why have variables stored in memory and accessible when in fact they are not needed?

Register Storage

- The storage-class specifier **register** can be placed before an automatic variable declaration
 - To suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers.

```
register int counter;
```
 - If intensely used variables such as counters or totals can be maintained in hardware registers
- **Often, register declarations are unnecessary**
 - Today's optimizing compilers are capable of recognizing frequently used variables
 - Can decide to place them in registers without the need for a register declaration

Static storage Classes

- Variables exist for entire program execution
- Default value of zero
- **static**: local variables defined in functions.
 - *Keep value after function ends*
 - Only known in their own function
- **extern**: default for *global variables* and functions
 - Known in any function

Tips for Storage Class

- **Defining a variable as global rather than local**
 - Allows unintended side effects to occur
 - When a function that does not need access to the variable accidentally or maliciously modifies it
- **In general, use of global variables** should be avoided : except in certain situations
- **Variables used only in a particular function**
 - Should be defined as local variables in that function
 - Rather than as external variables.

Scope Rules

- File scope
 - Identifier defined outside function, known in all functions
 - Used for *global variables, function definitions, function prototypes*
- Function scope
 - Can only be referenced inside a function body

Scope Rules

- Block scope
 - Identifier declared inside a block
 - Block scope begins at definition, ends at right brace
 - Used for *variables, function parameters (local variables of function)*
 - **Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block**
- Function prototype scope
 - Used for identifiers in *parameter list*

Scope Rule Example

```
int A; //global
int main() {
    A=1;
    MyProc();
    printf("A=%d\n", A);
    return 0;
}
```

```
void myProc() {
    int A=2;
    while(A==2) {
        int A=3;
        printf("A=%d\n", A);
        break;
    }
    printf("A=%d\n", A);
}
```

Outer blocks
"hidden" from inner blocks if there is a variable with the same name in the inner block

Printout:

A = 3

A = 2

A = 1

Scope and Life : Static Vs Global

```
int GA; //global
int main() {
    int i;
    GA=1;
    for(i=1;i<10;i++)
        MyProc();
    printf("GA=%d",GA);
    return 0;
}

void myProc() {
    static int SA=2;
    SA=SA+1;
}
```

Both SA and GA Variables exist for entire program execution

- SA initialized once
- SA can be accessible from myProc only
- But GA accessible from any part of Program

Scope Rule Example

```
int FunA(){return 4;}; //global
int main(){
```

```
{
  int FunA(){return 3;};
  printf("FA=%d\n", FunA());
}
```

```
printf("FA=%d\n", FunA());
return 0 ;
}
```

Outer blocks
"hidden" from inner
blocks if there is a
variable with the
same name in the
inner block

Printout:

FA = 3

FA = 4

Compile using gcc

This code will not compile
using c++/g++ compiler

Memory layout of C program

Dynamic memory allocation

- Reduce wastage of memory
- Useful when data size is unknown before hand
- Array Declaration

```
int A[100];
```

 - **Easy, Not to use pointer**, small size, known before
- Array Creation:
 - Not easy, use of pointer, typecast, **lager size, necessary size**

Memory management C: APIs

- Application program interfaces (APIs)
- Available function/APIs to manage memory
- Create/allocate/reserve space
 - malloc : memory allocation
 - calloc : memory allocation + initialization to 0
- Move a reserved space to another location
 - realloc: move the space to another location
- Destroy/de-allocate/free space
 - free:

Memory Allocation

- Memory can be allocated
- Declaring a variable

```
int A[100];
```

- Explicitly requesting space

```
int *A;
```

```
A = (int*) malloc(sizeof(int) * 100);
```

Example: Dynamic Array Allocation

- Given N persons (with their IQ level) in order
 - N may be dynamic, variable
- A person decide He/She is intelligent or dumb
- Decides locally:
 - If his/her IQ level is greater than equal to average of IQ level of both neighbors
 - Left neighbor and right neighbor

Example: Dynamic Array Allocation

```
main() {  
    int *IQScore, *Intelligent, i, N;  
    printf("Input N:"); scanf("%d", &N);  
    IQScore=(int*)malloc(N*sizeof(int));  
    Intelligent =(int*)calloc(N*sizeof(int));  
    for(i=0;i<N;i++)    scanf("%d",&IQScore[i]);  
    for(i=1;i<N-1;i++){  
        if(IQScore[i]>=(IQScore[i-1]+IQScore[i+1])/2)  
            Intelligent[i]=1; else Intelligent[i]=0;  
        printf(" I am %d person  %s\n", i,  
            Intelligent[i]?"YES":"NO");  
    }  
    free(IQScore);    free(Intelligent);  
}
```

Memory layout of C program

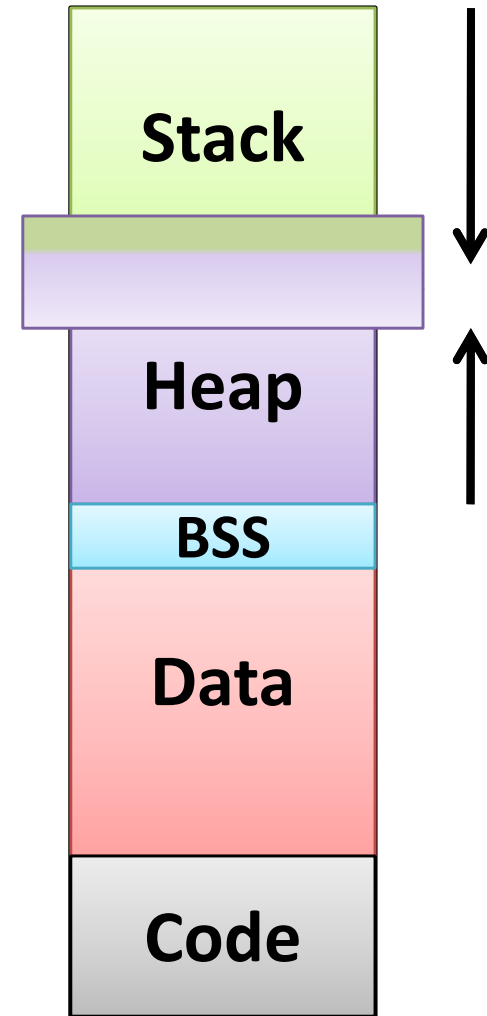
- Program: Input, Output, Processing
- Code (Instruction), Data (Stack, Heap)
- To store: Require memory
 - Input data, output data, intermediate data
- Memory can be allocated
 - Declaring a variable
 - Explicitly requesting space

```
int A[100];
```

```
int *A;  
A = (int*) malloc(sizeof(int)*100);
```

Memory layout of C program

- Stack
 - automatic (default), local
 - Initialized/uninitialized
- Data
 - Global, static, extern
 - BSS: Block Started by Symbol
 - BBS: Uninitialized Data Seg.
- Code : program instructions
- Heap
 - malloc, calloc



Memory layout of C program

```
int A;
```

```
int B=10;
```

```
main() {
```

```
    int Alocal;
```

```
    int *p;
```

```
    p= (int*) malloc(40);
```

```
}
```

Stack

Heap

BSS

Data

Code

```
$gcc test.c
```

```
$size a.out
```

text	data	bss	dec	hex	filename
1200	544	8	1752	6d8	a.out

Examples of Modular code using Functions

Modular Code for : X^n

```
#include <stdio.h>

int GetNum();
int PowXtoN(int n, int x);
void DispRes(int, int, int);

int main(){
    int X, N, Res;

    X=GetNum();
    N=GetNum();
    Res= PowXtonN(X,N);
    DispRes(Res,X,N);

    return 0;
}
```

```
int PowXtoN(int x, int n){
    int n, x, P=1, PS=x;
    while(n > 0) {
        if ((n%2)==1) P=P*PS;
        n=n/2; PS = PS* PS;
    }
    return P;
}
```

```
int GetANum(){ int N;
    printf("Enter a Number:");
    scanf("%d", &N);
    return N; }
```

```
void DispRes(int Res,
              int X, int N){
    printf("%d to power %d=%d",
           X, N, Res); }
```

Modular C Code : Square root of a Positive Number

```
#include <stdio.h>
float GetAccuracy();
float GetPosNum();
float SqRoot(float, float);
void DispRes(float, float);

int main(){
    float a, e, Res;

    a=GetPosNum();
    e=GetAccuracy();
    Res= SqRoot(a,e);
    DispRes(a, Res);

    return 0;
}
```

```
float SqRoot(float m,
              float e){

    float r1, r2;
    r1=m/2;
    r2=r1;
    while(abs(r1-r2)>e){
        r1=r2;
        r2=(r1+m/r1)/2;
    }
    return r2;
}
```

You need to write code of other functions

Modular C Code : Factorial of a Number

```
#include <stdio.h>

int GetNum();
int Factorial(int A);
void DispRes(int,int);

int main(){
    int a, Res;

    a=GetNum();
    Res= Factorial(a);
    DispRes(a, Res);

    return 0;
}
```

```
int Factorial(int n) {

    int Prod=1,i;
    for (i=1;i<=N;i++){
        Prod=Prod*i;
    }

    return prod;
}
```

You need to write code of other functions

Modular C Code : Reverse a Number

```
#include <stdio.h>

int GetNum();
int Reverse(int A);
void DispRes(int,int);

int main(){
    int a, Res;

    a=GetNum();
    Res= Reverse(a);
    DispRes(a, Res);

    return 0;
}
```

```
int Reverse(int n) {

    int RevNum=0, Rem;
    RevNum=0;
    while(n != 0) {
        Rem = n%10;
        RevNum=RevNum*10+ Rem;
        n=n/10;
    }
    return RevNum;
}
```

You need to write code of other functions

Modular C Code : Binary Search

```
#include <stdio.h>
int GetMinOfRange();
int GetMaxOfRange();
int GetUnknown();
int BinSrch(int,
            int, int);
void DispRes(int);
int main(){
    int Min, Max, X, Res;

    Min=GetMinOfRange();
    Max=GetMaxOfRange();
    X=GetUnknown();
    Res=BinSrch(Min,Max,X)
    DispRes(Res);
    return 0;
}
```

```
int BinSrch(int Rmin, int
            Rmax, int X){

    while (Rmin<Rmax){
        mid=(Rmin+Rmax)/2;
        if(X==mid) return mid;
        if (X>mid)
            Rmin=mid+1;
        else Rmax=mid;
    }
    return -1;
}
```

You need to write code of other functions

Modular C Code : Nth Fibonacci

```
#include <stdio.h>

int GetNum();
int Fib(int N);
void DispRes(int,int);

int main(){
    int N, Res;

    N=GetNum();
    Res= Fib(N);
    DispRes(N, Res);

    return 0;
}
```

```
int Fib(int N) {

    int fnm2=0; fnm1=1; n=2;
    if(N<=1) return 1;
    while(n<=N){
        fn = fnm2 + fnm1;
        fnm2=fnm1;
        fnm1=fn;
        n = n + 1;
    }
    return fn;
}
```

You need to write code of other functions

Modular C Code : GCD

```
#include <stdio.h>

int GetA();
int GetB();
int GCD(int A, int B);
Void DisRes(int,int,int);
int main(){
    int a, b, Res;

    a=GetA();
    b=GetB();
    Res= GCD(a,b);
    DisRes(a, b, Res);

    return 0;
}
```

```
int GCD(int n1, int n2) {
    while( !(n1==0 || n2==0) ) {
        if (n1>n2) n1=n1%n2;
        else n2=n2%n1;
    }
    if (n1==0) return n2;
    else return n1;
}
```

You need to write code of other functions

Modular Code Sin(x)

```
#include <stdio.h>
float GetX();
float GetAccuracy();
float SinXCal(float x,
             float acc);
void DispRes(float, float);

int main(){
    float X, acc, Res;

    X=GetX();
    acc=GetAccuracy();
    Res= SinXCal(X, acc);
    DispRes(X, Res);

    return 0;
}
```

```
float SinXCal(float x,
             float acc){
    int i=1;
    float SinXVal=0, term=x;
    while (term < acc) {
        i = i+2;
        term *= - x*x/(i*(i-1));
        SinxVal= SinxVal+ term;
    }
    return SinXVal;
}
```

You need to write code of other functions

Modular C Code : Value of PI

```
#include <stdio.h>

int GetABigNumer();
double ValPI(int N);
void
    DispRes(int, float);

int main(){
    int N;
    double Res;
    N=GetABigNumer();
    Res= ValPI(N);
    DispRes(a, b, Res);

    return 0;
}
```

```
double ValPI(int N){
    int M=0, i;
    double x, y, z, pi;
    for( i=0; i<N; i++) {
        x=(double)rand() / RAND_MAX;
        y=(double)rand() / RAND_MAX;
        z = x*x+y*y;
        if ( z<=1 ) M++;
    }
    pi=4.0*(double)M/N;
    return pi;
}
```

You need to write code of other functions

Modular C Code : Bisection Method

```
#include <stdio.h>
float GetMinOfInterVal();
float GetMaxOfInterVal();
float GetAccuracy();
float F(float a);

float Bisection(float a,
               float a, float acc)
void DispRes(float, float, float);
int main(){
    float a, b, acc, Res;
    a=GetMinOfInterVal();
    b=GetMaxOfInterVal();
    acc=GetAccuracy();
    Res= Bisection(a,b,acc);
    DispRes(a,b,Res);
    return 0;
}
```

```
float F(float a){
    return a*a*a-2*a-5;
}
```

```
float Bisection(float a,
               float a, float acc){
    float Fa, Fb, Fx;
    Fa=F(a); Fb=F(b);
    x=a; x1=b;
    while( abs(x-x1)>accuracy) {
        x1=x; x=(a+b)/2;
        Fx=F(x);
        if(Fa*Fx<0) b=x; else a=x;
    }
    return x;
}
```

You need to write code of other functions

Thanks