Chapter 6: Function & Recursion

- In this chapter, you will learn about
  - Introduction to function
  - User define function
    - Function prototype (declaration)
    - Function definition
    - Function call and return
  - Formal and Actual Parameters
  - Local and Global Variables
  - Storage classes
  - Recursion

Introduction

- A function is a block of code which is used to perform a specific task.
- It can be written in one program and used by another program without having to rewrite that piece of code. Hence, it promotes usability!!!
- Functions can be put in a library. If another program would like to use them, it will just need to include the appropriate header file at the beginning of the program and link to the correct library while compiling.

Introduction

- Functions can be divided into two categories :
  - Predefined functions (standard functions)
    - Built-in functions provided by C that are used by programmers without having to write any code for them. i.e: printf( ), scanf( ), etc
  - User-Defined functions
    - Functions that are written by the programmers themselves to carry out various individual tasks.

Standard Functions

- Standard functions are functions that have been pre-defined by C and put into standard C libraries.
  - Example: printf(), scanf(), pow(), ceil(), rand(), etc.
- What we need to do to use them is to include the appropriate header files.
  - Example: #include <stdio.h>, #include <math.h>
- What contained in the header files are the prototypes of the standard functions. The function definitions (the body of the functions) has been compiled and put into a standard C library which will be linked by the compiler during compilation.

User Defined Functions

- A programmer can create his/her own function(s).
- It is easier to plan and write our program if we divide it into several functions instead of writing a long piece of code inside the main function.
- A function is **<u>reusable</u>** and therefore prevents us (programmers) from having to unnecessarily rewrite what we have written before.
- In order to write and use our own function, we need to do the following:
  - create a function prototype (declare the function)
  - define the function somewhere in the program (implementation)
  - call the function whenever it needs to be used

Function Prototype or declaration

- A function prototype will tell the compiler that there exist a function with this name defined somewhere in the program and therefore it can be used even though the function has not yet been defined at that point.
- Function prototypes need to be written at the beginning of the program.
- If the function receives some arguments, the variable names for the arguments are not needed. Only the data type need to be stated.

Function Prototype or declaration cont…

- Function prototypes can also be put in a header file.Header files are files that have a **.h** extension.
- The header file can then be included at the beginning of our program.
- To include a user defined header file, type:

```
#include "header_file.h"
```

- Notice that instead of using < > as in the case of standard header files, we need to use " ". This will tell the compiler to search for the header file in the same directory as the program file instead of searching it in the directory where the standard library header files are stored.

Function Definitions

- Is also called as *function implementation*
- A function definition is where the actual code for the function is written. This code will determine what the function will do when it is called.
- A function definition consists of the following elements:
  - A function return data type (return value)
  - A function name
  - An optional list of formal parameters enclosed in parentheses (function arguments)
  - A compound statement (function body)

Function Definition

- A function definition has this format:

```
  return_data_type FunctionName (data_type variable1,
data_type variable2, data_type variable3, …..)
  {
        local variable declarations;
        statements;
  }
```

- The return data type indicates the type of data that will be returned by the function to the calling function. There can be only one return value.
- Any function not returning anything must be of type 'void'.
- If the function does not receive any arguments from the calling function, 'void' is used in the place of the arguments.

Function definition example 1

- A simple function is :

```
void print_message (void)
{
  printf("Hello, are you having fun?\n");
}
```

- Note the function name is **print_message**. No arguments are accepted by the function, this is indicated by the keyword **void** enclosed in the parentheses. The return_value is **void**, thus data is not returned by the function.
- So, when this function is called in the program, it will simply perform its intended task which is to print

  Hello, are you having fun?

Function definition example 1

- Consider the following example:

```
int calculate (int num1, int num2)
{
 int sum;
 sum = num1 + num2;
 return sum;
}
```

- The above code segments is a function named **calculate**. This function accepts two arguments i.e. **num 1** and **num2** of the type **int**. The return_value is **int**, thus this function will return an integer value.
- So, when this function is called in the program, it will perform its task which is to **calculate the sum of any two numbers** and **return the result of the summation.**
- Note that if the function is returning a value, it needs to use the keyword `return`.

Function Call

- Any function (including main) could utilize any other function definition that exist in a program – hence it is said to call the function (function call).
- To call a function (i.e. to ask another function to do something for the calling function), it requires the FunctionName followed by a list of actual parameters (or arguments), if any, enclosed in parenthesis.

Function Call cont…

- If the function requires some arguments to be passed along, then the arguments need to be listed in the bracket ( ) according to the specified order. For example:

```
void Calc(int, double, char, int);

int main(void)
{
        int a, b;
        double c;
        char d;
        …
        Calc(a, c, d, b);
    return (0);
}
```

Function Call

Function Call cont…

- If the function returns a value, then the returned value need to be assigned to a variable so that it can be stored. For example:

```
int GetUserInput (void); /* function prototype*/
int main(void)
{
     int input;
     input = GetUserInput( );
     return(0);
}
```

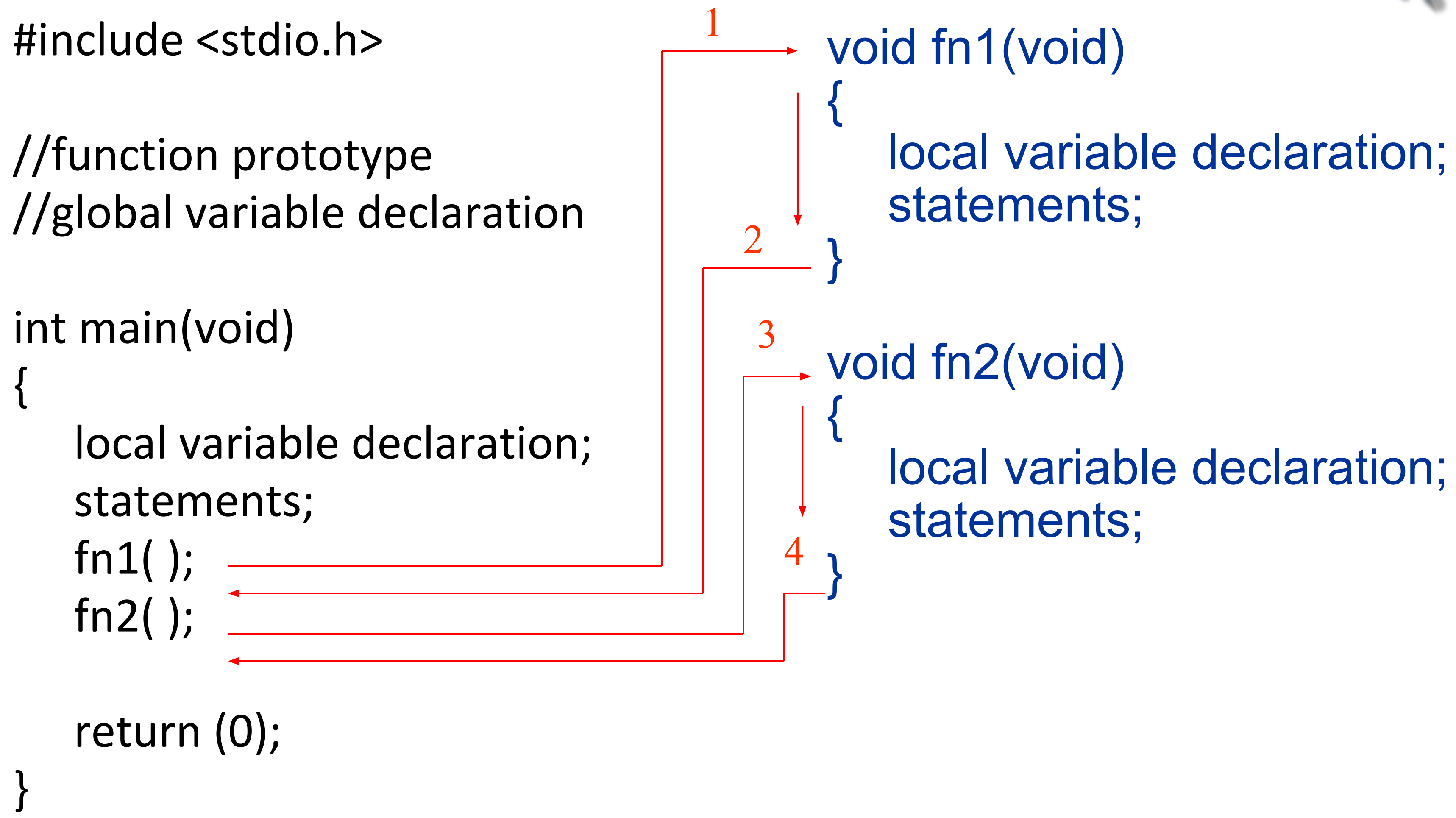
- However, it is perfectly okay (syntax wise) to just call the function without assigning it to any variable if we want to ignore the returned value.
- We can also call a function inside another function. For example:

```
printf("User input is: %d", GetUserInput( ));
```

Function call cont…

- There are 2 ways to call a function:
  - Call by value
    - In this method, only the **copy of variable's value** (copy of actual parameter's value) is passed to the function. Any modification to the passed value inside the function **will not** affect the actual value.
    - In all the examples that we have seen so far, this is the method that has been used.
  - Call by reference
    - In this method, the **reference** (memory address) of the variable is passed to the function. Any modification passed done to the variable inside the function **will** affect the actual value.
    - To do this, we need to have knowledge about pointers and arrays, which will not be discussed in this course.

Basic skeleton…

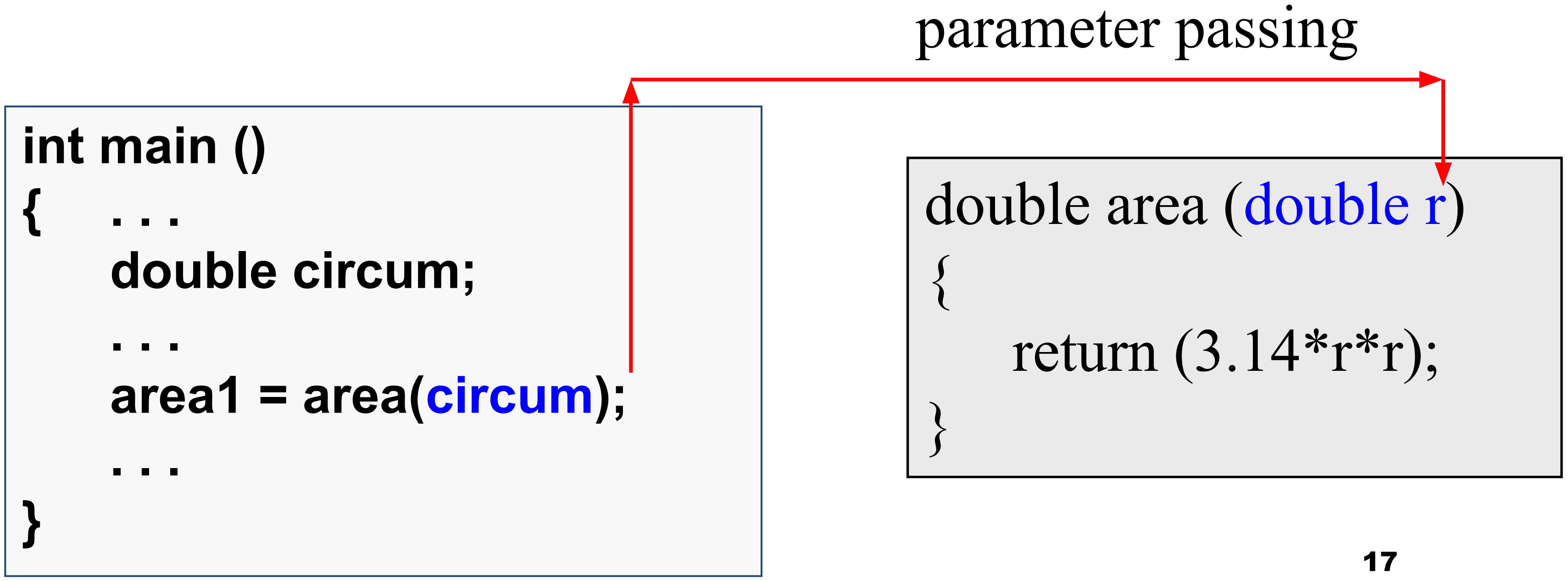


```
#include <stdio.h>

//function prototype
//global variable declaration

int main(void)
{
    local variable declaration;
    statements;
    fn1( );
    fn2( );

    return (0);
}
```

```
void fn1(void)
{
    local variable declaration;
    statements;
}

void fn2(void)
{
    local variable declaration;
    statements;
}
```

Parameter passing

- When the function is executed, the value of the actual parameter is copied to the formal parameter



```
int main ()
{    . . .
     double circum;
     . . .
     area1 = area(circum);
     . . .
}
```

```
double area (double r)
{
     return (3.14*r*r);
}
```

Example of function definition

Return-value type

Formal parameters

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
    }
    return (A);
}
```

BODY

Value returned

Return value

- A function can return a value
  - Using **return** statement
- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
int x, y, z;
scanf("%d%d", &x, &y);
z = gcd(x,y);
printf("GCD of %d and %d is %d\n", x, y, z);
```

Function Not Returning Any Value

- Example: A function which prints if a number is divisible by 7 or not

```
void  div7 (int n)
{
   if  ((n % 7) == 0)
       printf (“%d is divisible by 7”, n);
   else
       printf (“%d is not divisible by 7”, n);
   return;
}
```

Return type is void

Optional

return statement

- In a value-returning function (return type is not void), return does two distinct things
  - specify the value returned by the execution of the function
  - terminate that execution of the callee and transfer control back to the caller
- A function can only return one value
  - The value can be any expression matching the return type
  - but it might contain more than one return statement.
- In a void function
  - return is optional at the end of the function body.
  - return may also be used to terminate execution of the function explicitly.
  - No return value should appear following return.

```
void compute_and_print_itax ()
{
      float income;
      scanf ("%f", &income);
      if (income < 50000) {
        printf ("Income tax = Nil\n");
        return;
      }
      if (income < 60000) {
        printf ("Income tax = %f\n", 0.1*(income-50000));
        return;
      }
      if (income < 150000)     {
         printf ("Income tax = %f\n", 0.2*(income-60000)+1000);
        return ;
      }
      printf ("Income tax = %f\n", 0.3*(income-150000)+19000);
}
```

Terminate function execution before reaching the end

Calling function (Caller)

Called function (Callee)

parameter

```
void main()
{ float cent, fahr;
  scanf(“%f”,&cent);
  fahr = cent2fahr(cent);
  printf(“%fC = %fF\n”,
    cent, fahr);
}
```

```
float cent2fahr(float data)
{
  float result;
  result = data*9/5 + 32;
  return result;
}
```

Parameter passed

Returning value

Calling/Invoking the cent2fahr function

How it runs

```
float cent2fahr(float data)
{
  float result;
  printf(“data = %f\n”, data);
  result = data*9/5 + 32;
  return result;
   printf(“result = %f\n”, result);
}
void main()
{ float cent, fahr;
  scanf(“%f”,&cent);
  printf(“Input is %f\n”, cent);
  fahr = cent2fahr(cent);
  printf(“%fC = %fF\n”, cent, fahr);
}
```

Output

```
$ ./a.out
32
Input is 32.000000
data = 32.000000
32.000000C = 89.599998F

$./a.out
-45.6
Input is -45.599998
data = -45.599998
-45.599998C = -50.079998F
$
```

Function Calling – Different Aspects

The functions available in a code may accept an argument or may not accept it at all. Thus, it may return any of the values or may not do so. On the basis of these facts, the function calls have the following four aspects:

- A function that has no arguments and has no return value.
- A function that has no arguments but has a return value.
- A function that has arguments but has no return value.
- A function that has arguments and also has a return value.

Example 1: receive nothing and return nothing

```c
#include <stdio.h>
void greeting(void); /* function prototype */

int main(void)
{
   greeting( );
   greeting( );
   return(0);
}

void greeting(void)
{
   printf("Have fun!! \n");
}
```

In this example, function greeting does not receive any arguments from the calling function (main), and does not return any value to the calling function, hence type ‘void’ is used for both the input arguments and return data type.

```
Have fun!!
Have fun!!
Press any key to continue
```

Example 2: receive nothing and return a value

```c
#include <stdio.h>
int getInput(void);

int main(void)
{
   int num1, num2, sum;

   num1 = getInput( );
   num2 = getInput( );
   sum = num1 + num2;

   printf("Sum is %d\n",sum);
   return(0);
}

int getInput(void)
{
   int number;
   printf("Enter a number:");
   scanf("%d“,&number);

   return number;
}
```

```
Enter a number: 5
Enter a number: 4
Sum is 9
Press any key to continue
```

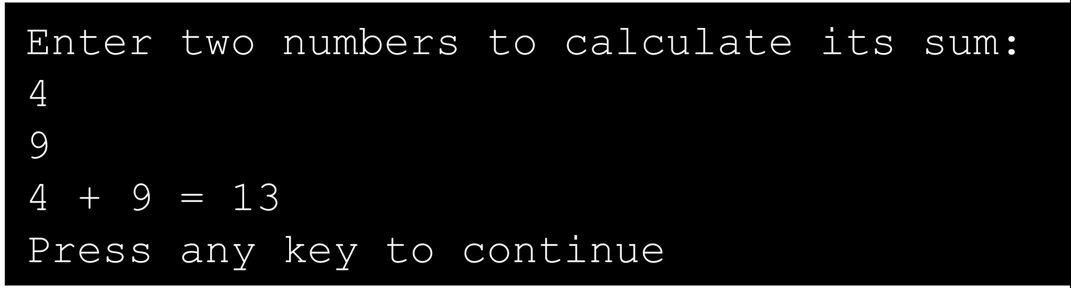
Example 3: receive parameter(s) and return nothing

```c
#include <stdio.h>
int getInput(void);
void displayOutput(int);

int main(void)
{
    int num1, num2, sum;

    num1 = getInput();
    num2 = getInput();
    sum = num1 + num2;
    displayOutput(sum);
    return(0);
}

int getInput(void)
{
    int number;
    printf("Enter a number:");
    scanf("%d",&number);

    return number;
}

void displayOutput(int sum)
{
    printf("Sum is %d \n",sum);
}
```

```
Enter a number: 5
Enter a number: 4
Sum is 9
Press any key to continue
```

Example 4: receive parameters and return a value

```
#include <stdio.h>
int calSum(int,int);        /*function prototype*/

int main(void)
{
   int sum, num1, num2;

   printf("Enter two numbers to calculate its sum:\n");

   scanf("%d%d",&num1,&num2);

   sum = calSum(num1,num2);  /* function call */
   printf("\n %d + %d = %d", num1, num2, sum);

   return(0);
}
int calSum(int val1, int val2)  /*function definition*/
{
   int sum;
   sum = val1 + val2;
   return sum;
}
```



```
Enter two numbers to calculate its sum:
4
9
4 + 9 = 13
Press any key to continue
```

Example 4 explanation

- In this example, the calSum function receives input parameters of type int from the calling function (main).
- The calSum function returns a value of type int to the calling function.
- Therefore, the **function definition** for calSum:

  int calSum(int val1, int val2)
- Note that the **function prototype** only indicates the type of variables used, not the names.

  int calSum(int,int);
- Note that the function call is done by (main) calling the function name (calSum), and supplying the variables (num1,num2) needed by the calSum function to carry out its task.

Example of function definition

Return-value type

Formal parameters

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
    }
    return (A);
}
```

BODY

Value returned

Return value

- A function can return a value
  - Using **return** statement
- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
int x, y, z;
scanf("%d%d", &x, &y);
z = gcd(x,y);
printf("GCD of %d and %d is %d\n", x, y, z);
```

FUNCTIONS WITH THE ARGUMENTS, NO RETURN VALUES
Program

```
/* Function declaration */
void printline (void);
void value (void);
 main()
 {
 printline();
 value();
 printline();
 }

 /* Function1: printline( ) */
 void printline(void) /* contains no arguments */
 {
 int i ;
 for(i=1; i <= 35; i++)
 printf("%c",'-');
 printf("\n");
 }
 /* Function2: value( ) */
 void value(void) /* contains no arguments */
 {
 int year, period;
 float inrate, sum, principal;
 printf("Principal amount?");
 scanf("%f", &principal);
 printf("Interest rate? ");
 scanf("%f", &inrate);
 printf("Period? ");
 scanf("%d", &period);
 sum = principal;
 year = 1;
 while(year <= period)
 {
 sum = sum *(1+inrate);
 year = year +1;
 }
 printf("\n%8.2f %5.2f %5d %12.2f\n",
 principal,inrate,period,sum);
 }
```

Output

```
-----------------------------------
 Principal amount? 5000
 Interest rate? 0.12
 Period? 5

 5000.00 0.12 5 8811.71
```

FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUES
Program

```
/* prototypes */
void printline (char c);
void value (float, float, int);

main( )
{
float principal, inrate;
int period;

printf("Enter principal amount, interest");
printf(" rate, and period \n");
scanf("%f %f %d",&principal, &inrate, &period);
printline('Z');
value(principal,inrate,period);
printline('C');
}
void printline(char ch)
{
int i ;
for(i=1; i <= 52; i++)
printf("%c",ch);
printf("\n");
}
```

```
void value(float p, float r, int n)
{
int year ;
float sum ;
sum = p ;
year = 1;
while(year <= n)
{
sum = sum * (1+r);
year = year +1;
}
printf("%f\t%f\t%d\t%f\n",p,r,n,sum);
}
```

Output

```
Enter principal amount, interest rate, and period
5000 0.12 5
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
5000.000000 0.120000 5 8811.708984

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCC
```

FUNCTIONS WITH ARGUMENTS AND RETURN VALUES
Program

```
void printline (char ch, int len);
value (float, float, int);
main( )
{
float principal, inrate, amount;
int period;
printf(“Enter principal amount, interest”);
printf(“rate, and period\n”);
scanf(%f %f %d”, &principal, &inrate, &period);
printline (‘*’ , 52);
amount = value (principal, inrate, period);
printf(“\n%f\t%f\t%d\t%f\n\n”,principal,
inrate,period,amount);
printline(‘=’,52);
}
void printline(char ch, int len)
{
int i;
for (i=1;i<=len;i++) printf(“%c”,ch);
printf(“\n”);
}
```

```
value(float p, float r, int n) /* default return type */
{
int year;
float sum;
sum = p; year = 1;
while(year <=n)
{
sum = sum * (l+r);
year = year +1;
}
return(sum); /* returns int part of sum */
}
```

Output

```
Enter principal amount, interest rate, and period
5000 0.12 5
**************************************************
5000.000000 0.1200000 5 8811.000000
= = = = = = = = = = = = = = = = = = = = = = = = = =
```

Write a function power that computes x raised to the power y for integers x and y and returns

```
main( )
{ int x,y; /*input data */
double power(int, int); /* prototype declaration*/
printf("Enter x,y:");
scanf("%d %d" , &x,&y);
printf("%d to power %d is %f\n", x,y,power (x,y));
}
double power (int x, int y);
{
double p;
p = 1.0 ; /* x to power zero */
if(y >=o)
while(y--) /* computes positive powers */
p *= x;
else
while (y++) /* computes negative powers */
p /= x;
return(p);
}
```

Output

```
Enter x,y:16 2
16 to power 2 is 256.000000
Enter x,y:16 -2
16 to power -2 is 0.003906
```

Example
Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

```
#include <math.h>
 #define SIZE 5
 float std_dev(float a[], int n);
 float mean (float a[], int n);
 main( )
 {
 float value[SIZE];
 int i;
 printf("Enter %d float values\n", SIZE);
 for (i=0 ;i < SIZE ; i++)
 scanf("%f", &value[i]);
 printf("Std.deviation is %f\n", std_dev(value,SIZE));
 }
 float std_dev(float a[], int n)
 { int i;
 float x, sum = 0.0;
 x = mean (a,n);
 for(i=0; i < n; i++)
 sum += (x-a[i])*(x-a[i]);
 return(sqrt(sum/(float)n));
 }
```

```
float mean(float a[],int n)
 {
 int i ;
 float sum = 0.0;
 for(i=0 ; i < n ; i++)
 sum = sum + a[i];
 return(sum/(float)n);
 }
```

Output

```
Enter 5 float values
 35.0 67.0 79.5 14.20 55.75

 Std.deviation is 23.231582
```

Write a program that uses a function to sort an array of integers.

```
void sort(int m, int x[ ]);
 main()
 {
 int i;
 int marks[5] = {40, 90, 73, 81, 35};

 printf("Marks before sorting\n");
 for(i = 0; i < 5; i++)
 printf("%d ", marks[i]);
 printf("\n\n");

 sort (5, marks);
 printf("Marks after sorting\n");
 for(i = 0; i < 5; i++) printf("%4d", marks[i]);
 printf("\n");
 }
```

```
void sort(int m, int x[ ])
{
int i, j, t;
for(i = 1; i <= m-1; i++)
for(j = 1; j <= m-i; j++)
if(x[j-1] >= x[j])
{
t = x[j-1];
x[j-1] = x[j];
x[j] = t;
}
}
```

OUTPUT

```
Marks before sorting
 40 90 73 81 35

 Marks after sorting
 35 40 73 81 90
```

Function Prototypes

- Function prototypes are usually written at the beginning of a program, ahead of any functions (including main())
- Prototypes can specify parameter names or just types (more common)
- Examples:

```
int  gcd (int , int );
void div7 (int number);
```

  - Note the semicolon at the end of the line.
  - The parameter name, if specifed, can be anything; but it is a good practice to use the same names as in the function definition

Another Example

Function declaration (prototype)

```
int  factorial (int m);
```

```
int main()
{
    int  n;
    for  (n=1; n<=10; n++)
     printf (“%d! = %d \n”,
                  n, factorial (n) );
}
```

Function call

```
int  factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
     temp = temp * i;
    return (temp);
}
```

Function definition

Output

```
1! = 1
2! = 2
3! = 6  …….. upto 10!
```

Calling a function

- Called by specifying the function name and parameters in an instruction in the calling function
- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types

Example

Formal parameters

```
double operate (double x, double y, char op)
{
     switch (op) {
          case '+' : return x+y+0.5 ;
          case '~'  : if (x>y)
                    return x-y + 0.5;
                      return y-x+0.5;
          case 'x' : return x*y + 0.5;
          default : return –1;
     }
}
```

```
void main ()
{
     double x, y, z;
     char op;
     . . .
    z = operate (x, y, op);
      . . .
}
```

Actual parameters

Local variables

Calling function (Caller)

Called function (Callee)

parameter

```
void main()
{ float cent, fahr;
  scanf("%f",&cent);
  fahr = cent2fahr(cent);
  printf("%fC = %fF\n",
    cent, fahr);
}
```

```
float cent2fahr(float data)
{
  float result;
  result = data*9/5 + 32;
  return result;
}
```

Parameter passed

Returning value

Calling/Invoking the cent2fahr function

Local variables

- A function can define its own local variables
- The locals have meaning only within the function
  - Each execution of the function uses a new set of locals
  - Local variables cease to exist when the function returns
- Parameters are also local

Local variables

```
/* Find the area of a circle with diameter d */
double circle_area (double d)
{
    double radius, area;
    radius = d/2.0;
    area = 3.14*radius*radius;
    return (area);
}
```

parameter

local variables

Points to note

- The identifiers used as formal parameters are “local”.
  - Not recognized outside the function
  - Names of formal and actual arguments may differ
- A value-returning function is called by including it in an expression
  - A function with return type T (≠ void) can be used anywhere an expression of type T

- Returning control back to the caller
  - If nothing returned
    - return;
    - or, until reaches the last right brace ending the function body
  - If something returned
    - return expression;

Some more points

- A function cannot be defined within another function
  - All function definitions must be disjoint
- Nested function calls are allowed
  - A calls B, B calls C, C calls D, etc.
  - The function called last will be the first to return
- A function can also call itself, either directly or in a cycle
  - A calls B, B calls C, C calls back A.
  - Called recursive call or recursion

Example: main calls ncr, ncr calls fact

```
int ncr (int n, int r);
int fact (int n);

void main()
{
     int i, m, n, sum=0;
     scanf (“%d %d”, &m, &n);
     for (i=1; i<=m; i+=2)
         sum = sum + ncr (n, i);
     printf (“Result: %d \n”, sum);
}
```

```
int  ncr (int n, int r)
{
     return (fact(n) / fact(r) /
     fact(n-r));
}

int  fact (int n)
{
     int  i, temp=1;
     for (i=1; i<=n; i++)
         temp *= i;
     return (temp);
}
```

Scope of a variable

- Part of the program from which the value of the variable can be used (seen)
- Scope of a variable - Within the block in which the variable is defined
  - Block = group of statements enclosed within { }
- Local variable – scope is usually the function in which it is defined
  - So two local variables of two functions can have the same name, but they are different variables
- Global variables – declared outside all functions (even main)
  - scope is entire program by default, but can be hidden in a block if local variable of same name defined

Variable

```
#include  <stdio.h>
int A = 1;
void main()
    {
      myProc();
      printf ( "A = %d\n", A);
    }

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

Global variable

Hides the global A

Output:

```
A = 3

A = 1
```

Compute GCD of two numbers

```
int main() {
      int  A, B, temp;
      scanf ("%d %d", &A, &B);
      if  (A > B)  {
          temp = A;  A = B;  B = temp;
    }
      while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
      }
      printf ("The GCD is %d", A);
}
```

```
12 )  45  (  3
      36
      ----
        9  )  12  (  1
                9
               ---
                3 ) 9  (  3
                    9
                   ---
                    0
```

Initial: A=12, B=45
Iteration 1: temp=9, B=12,A=9
Iteration 2: temp=3, B=9, A=3
B % A = 0 □ GCD is 3

gcd.c

Compute GCD of two numbers

```
int x, y, z;
scanf(“%d%d”, &x, &y);
z = gcd(x,y);
printf(“GCD of %d and %d is %d\n”, x, y, z);
```

Return-value type

Formal parameters

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
    }
    return (A);
}
```

Value returned

BODY

Function – complete flow

```
#include<stdio.h>
int calSum(int, int);
int main(void)
{
   int num1, num2, sum;
   printf(“Enter 2 numbers to calculate its sum:”);
   scanf(“%d %d”, &num1, &num2);

   sum = calSum (num1, num2);
   printf (“\n %d + %d = %d”, num1, num2, sum);
   return (0);
}

int calSum(int val1, int val2)
{
   int sum;
   sum = val1 + val2;
   return sum;
}
```

Function with parameter/argument

- When a function calls another function to perform a task, the calling function may also send data to the called function. After completing its task, the called function may pass the data it has generated back to the calling function.
- Two terms used:
  - Formal parameter
    - Variables declared in the formal list of the function header (written in function prototype & function definition)
  - Actual parameter
    - Constants, variables, or expression in a function call that correspond to its formal parameter

Function with parameter/argument…

- The three important points concerning functions with parameters are: (number, order and type)
  - The number of actual parameters in a function call must be the same as the number of formal parameters in the function definition.
  - A one-to-one correspondence must occur among the actual and formal parameters. The first actual parameter must correspond to the first formal parameter and the second to the second formal parameter, an so on.
  - The type of each actual parameter must be the same as that of the corresponding formal parameter.

Formal / Actual parameters

```
#include <stdio.h>
int calSum(int,int);                /*function prototype*/

int main(void)
{
    .....
    .....
    sum = calSum(num1,num2);  /* function call */
    .....
}

int calSum(int val1, int val2)  /*function header*/
{
    ......
    ......
    ......
}
```

Formal Parameters

Actual Parameters

Formal Parameters

Declaration of Variables

- Up until now, we have learnt to declare our variables within the braces of segments (or a function) including the main.

- It is also possible to declare variables outside a function. These variables can be <u>accessed by all functions</u> throughout the program.

Local and Global Variables

- Local variables only exist within a function. After leaving the function, they are ‘destroyed’. When the function is called again, they will be created (reassigned).
- Global variables can be accessed by any function within the program. While loading the program, memory locations are reserved for these variables and any function can access these variables for read and write (overwrite).
- If there exist a local variable and a global variable with the same name, the compiler will refer to the local variable.

Global variable - example

```c
#include <stdio.h>
void initialise(void);
void getInputs(void);
void calSum(void);

int sum, num1, num2;

int main(void)
{
    /* initialise sum to 0 */
    initialise( );

    /* read num1 and num2 */
    getInputs( );

    calSum( );

    printf("Sum is %d\n",sum);
    return (0);
}

void initialise(void)
{
    sum = 0;
}

void getInputs(void)
{
    printf("Enter num1 and num2:\n");
    scanf("%d%d",&num1,&num2);
}

void calSum(void)
{
    sum = num1 + num2;
}
```

Global variables

```
Enter num1 and num2:
4
9
Sum is 13
Press any key to continue
```

Global variable – example explained

- In the previous example, no variables are passed between functions.
- Each function could have access to the global variables, hence having the right to read and write the value to it.
- Even though the use of global variables can simplify the code writing process (promising usage), it could also be dangerous at the same time.
- Since any function can have the right to overwrite the value in global variables, a function reading a value from a global variable can not be guaranteed about its validity.

Global variable – the dangerous side

```c
#include <stdio.h>
void initialise(void);
void getInputs(void);
void calSum(void);

int sum, num1, num2;

int main(void)
{
    /* initialise sum to 0 */
    initialise( );

    /* read num1 and num2 */
    getInputs( );

    calSum( );

    printf("Sum is %d\n",sum);
    return(0);
}

void initialise(void)
{
    sum = 0;
}

void getInputs(void)
{
    printf("Enter num1 and num2:\n");
    scanf("%d%d",&num1,&num2);
}

void calSum(void)
{
    sum = num1 + num2;
    initialise();
}
```

```
Enter num1 and num2:
4
9
Sum is 0
Press any key to continue
```

Imagine what would be the output of this program if someone 'accidently' write the following function call inside calSum?

Storage Classes

- Local variables only exist within a function by default. When calling a function repeatedly, we might want to
  - Start from scratch – re-initialise the variables
    - The storage class is ‘auto’
  - Continue where we left off – remember the last value
    - The storage class is ‘static’
- Another two storage classes (seldomly used)
  - register (ask to use hardware registers if available)
  - extern (global variables are external)

Auto storage class

- Variables with automatic storage duration are created when the block in which they are declared is entered, exist when the block is active and destroyed when the block is exited.
- The keyword **auto** explicitly declares variables of automatic storage duration. It is rarely used because when we declare a **local variable**, by default it has class storage of type **auto**.
  - `int a, b;` is the same as `auto int a, b;`

Static storage class

- Variables with static storage duration exist from the point at which the program begin execution.
- All the global variables are static, and all local variables and functions formal parameters are automatic by default. Since all global variables are static by default, in general it is not necessary to use the **static** keyword in their declarations.
- However the **static** keyword can be applied to a local variable so that the variable still exist even though the program has gone out of the function. As a result, whenever the program enters the function again, the value in the **static** variable still holds.

Auto - Example

```c
#include <stdio.h>
void auto_example(void);

int main(void)
{
   int i;

   printf("Auto example:\n");

   auto_example( );
   auto_example( );
   auto_example( );

   return(0);

}

void auto_example(void)
{
   auto int num = 1;

   printf(“ %d\n”,num);
   num = num + 2;
}
```

```
Auto example:
1
1
1
Press any key to continue
```

Static - Example

```
#include <stdio.h>
void auto_example(void);

int main(void)
{
   int i;

   printf(“Static example:\n");

   static_example( );
   static_example( );
   static_example( );

   return(0);

}

void static_example(void)
{
   static int num = 1;

   printf(“ %d\n”,num);
   num = num + 2;
}
```

```
Static example:
1
3
5
Press any key to continue
```

Simple Recursion

- Recursion is where a function calls itself.
- Concept of recursive function:
  - A recursive function is called to solve a problem
  - The function only knows how to solve the simplest case of the problem. When the simplest case is given as an input, the function will immediately return with an answer.
  - However, if a more complex input is given, a recursive function will divide the problem into 2 pieces: a part that it knows how to solve and another part that it does not know how to solve.

Simple Recursion cont…

- The part that it does not know how to solve always resembles the original problem, but of a slightly simpler version.
- Therefore, the function calls itself to solve this simpler piece of problem that it does now know how to solve. This is called the recursion step.
- The recursion step is done until the problem converges to become the simplest case.
- This simplest case will be solved by the function which will then return the answer to the previous copy of the function.
- The sequence of *returns* will then go all the way up until the original call of the function finally return the result.

Simple Recursion cont…

- Any problem that can be solved recursively can also be solved iteratively (using loop).
- Recursive functions are slow and takes a lot of memory space compared to iterative functions
- So why bother with recursion? There are 2 reasons:
  - Recursion approach more naturally resembles the problem and therefore the program is easier to understand and debug.
  - Iterative solution might not be apparent.

Example 1 - x^y

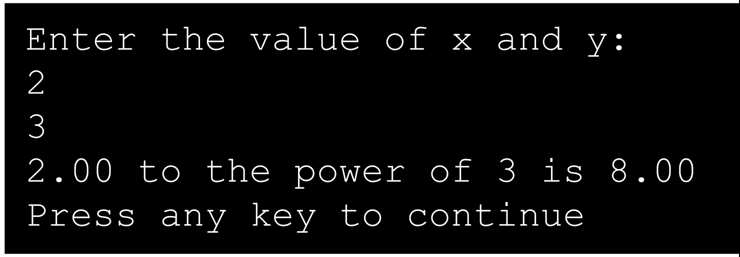
- In this example, we want to calculate x to the power of y (i.e. x^y)
- If we analyze the formula for x^y, we could see that x^y could be written as (x being multiplied to itself, y times).
- An example is 2^4, which can be written as

  $2^4 = 2 \times 2 \times 2 \times 2$ (in this case, x = 2, y = 4)
- 2^4 could also be rewritten as

  $2^4 = 2^1 \times 2^3$ where $2^1 = 2$ (i.e the number itself)
- Therefore, we could divide the problem into two stage:
  - Simplest case: when y = 1, the answer is x
  - Recursive case, we need to solve for $x * x^{(y-1)}$

Example 1 - x^y

```
#include <stdio.h>
double XpowerY(double,int);

int main(void)
{
   double power, x;
   int y;

   printf("Enter the value of x and y:\n");
   scanf("%lf%d",&x,&y);

   power = XpowerY(x,y);

   printf("%.2f to the power of %d is %.2f\n\n",x,y,power);
   return(0);
}

double XpowerY(double x, int y)
{
   if (y ==1 )
    return x;
   else
    return x * XpowerY(x, y-1);
}
```



```
Enter the value of x and y:
2
3
2.00 to the power of 3 is 8.00
Press any key to continue
```

Example 2 - Factorial

- Analysis:
  - n!= n * (n-1) * (n-2) * (n-3) * (n-4) ……… * 1
  - n! could be rewritten as n * (n-1)!
  - Example:
    - 5! = 5 * 4 * 3 * 2 * 1 = 5 * (4)!, where n = 5
  - Fact: 0! Or 1! is equal to 1
- Therefore, we could divide this problem into two stages for n!:
  - Simplest case: if (n <= 1), answer is 1
  - Recursive case: we need to solve for n * (n-1)!

Example 2 - Factorial

```c
#include <stdio.h>
double fact(double);

int main(void)
{
   double n, result;

   printf("Please enter the value of n:");
   scanf("%lf",&n);

   result = fact(n);

   printf(" %.f! = %.2f\n",n,result);

   return(0);
}

double fact(double n)
{
   if (n <= 1)
    return 1;
   else
    return n * fact(n-1);
}
```

```
Enter the value of n: 3
3! = 6.00
Press any key to continue
```

Summary

- In this chapter, you have learnt:
  - Standard vs User Define functions
  - Function prototype, function definition and function call
  - Formal vs Actual parameters
  - Local vs Global variables
  - Auto vs Static storage class
  - Simple recursion