PROGRAMMING FOR PROBLEM SOLVING (CS103ES)

UNIT - IV: Function and Dynamic Memory Allocation

Functions: Designing structured programs, Declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value, Passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions **Dynamic memory allocation:** Allocating and freeing memory, Allocating memory for arrays of different data types

FUNCTIONS:

- ✓ The programs we have presented so far have been very simple. They solved problems that could be understood without too much effort.
- ✓ As the program size grows and it is common practice to divide the complex program into smaller elementary parts.
- ✓ The planning for large programs is simpler. First, we must understand the problem as a whole; then break it into simpler and smaller understandable parts. We call each of these parts of a program a module and subdividing a problem into manageable parts **top-down design.**
- ✓ The technique used to pass data to a function is known as **parameter passing.**

Functions in C:

- ✓ In C, the idea of top-down design is done using functions. A C program is made of one or more functions, but one and only one of which must be named main.
- ✓ The execution of the program always starts and ends with main, but it can call other functions to do some of the job.
- ✓ A function in main including main is an independent module that will be called to do a specific task. A called function receives control from calling function.
- ✓ When the called function completes its task, it returns control back to the calling function. It may or may not return a value to the caller.
- ✓ The function main is called by the operating system; main in turn calls other functions. When main is complete, control return to the operating system.

Mr. Mohammed Afzal, Asst. Professor in AIML
Mob: +91-8179700193, Email: mdafzal.mbnr@gmail.com

- ✓ In general, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.

Advantages of using functions in C:

- ✓ Problems can be divided into understandable and manageable steps.
- ✓ Functions provide a way to reuse code that can be used in more than one place in the program.
- ✓ Functions are used to protect data. The local data described in a function is available only to the function and only while the function is executing. Local data in a function cannot be seen or changed by a function outside of its scope.

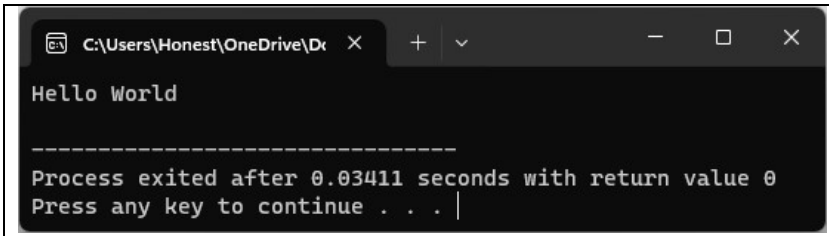
User-Defined Functions:

- ✓ Like every other object in C, functions must be declared, defined and called when needed.
- ✓ The function declaration needs to done before the function call, mentions the name of the function, the return type and type and order of the formal parameters. The declaration uses only the header of the function and ended with a semicolon.
- ✓ The function definition can be coded after the function call contains the code needed to complete the task.
- ✓ The function call comes within a function which calls it to get the portion of the job to be done.
- ✓ A function name is used three times: for declaration, in a call and for definition.
- ✓ **Example programs:**

// Simple greetings program using functions

funDemo.c

```c
#include <stdio.h>

//Function Declaration
void greetings(void);

int main(void)
{
    //Function Call
    greetings();
    return 0;
}

//Function Definition
void greetings()
{
    printf("Hello World\n");
}
```



Basic Function Design:

- ✓ The classification of the basic function designs is done by their return values and their parameter list.
- ✓ Functions either return a value or don't. Functions that don't return a value are known as void functions.
- ✓ Based on requirement parameter list is also optional. Either they can have parameters or don't.
- ✓ Combining the return values and parameter lists functions can be classified into four basic designs:
  1. void functions without parameters
  2. void functions with parameters
  3. non void functions without parameters
  4. non void functions with parameters

1. void functions without parameters:

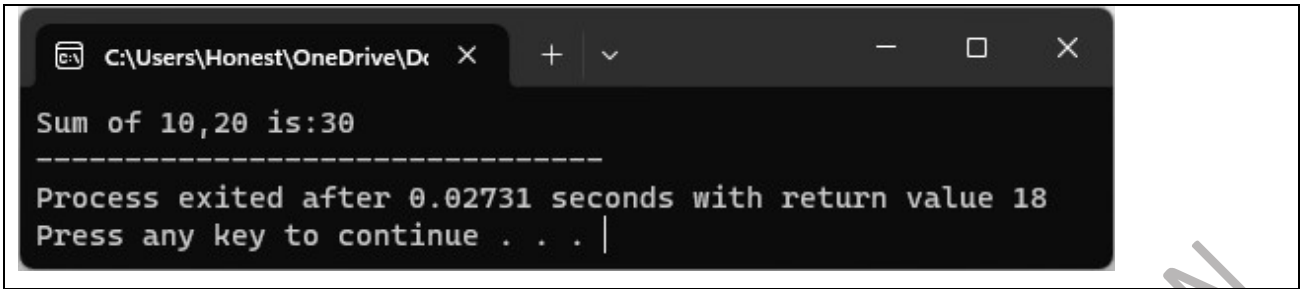
- ✓ A void function can be written without any parameters.
- ✓ As the void function does not have a return value, it can be used only as a statement. It cannot be used in an expression.

result = greeting(); // error. void function

// void functions without parameters

voidWOPar.c

```c
#include <stdio.h>
//Function Declaration
void add(void);
int main(void)
{
    //Function Call
    add();
    return 0;
}

//Function Definition
void add(void)
{
    int a=10,b=20,sum;
    sum=a+b;
    printf("Sum of %d,%d is:%d",a,b,sum);
}
```



2. void function with parameters

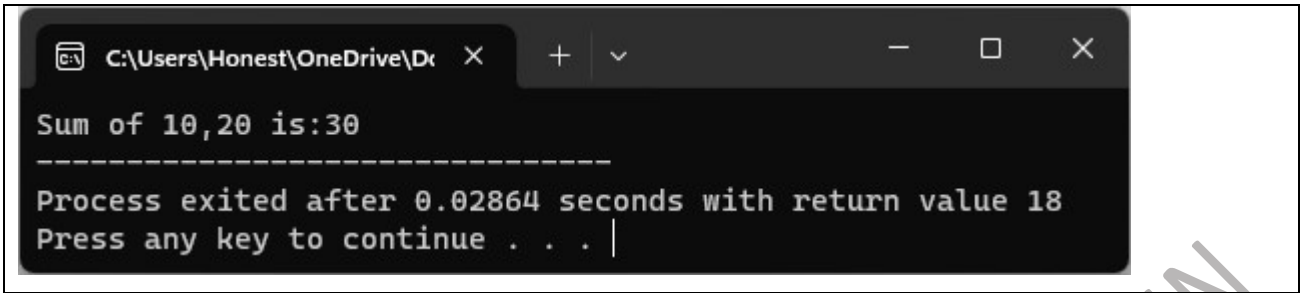
- ✓ A void function can have parameters.
- ✓ Parameters are helpful in passing the values from the calling function to the called function. Passing Parameters, is a technique of sending the values to the called function.
- ✓ As the void function does not have a return value, it can be used only as a statement. It cannot be used in an expression.

// void functions with parameters

voidWPar.c

```c
#include <stdio.h>
//Function Declaration
void add(int,int);

int main(void)
{
    int a=10,b=20;

    //Funciton Call
    add(a,b);
    return 0;
}

//Fucntion Definition
void add(int x, int y)
{
    int sum;
    sum=x+y;
    printf("Sum of %d,%d is:%d",x,y,sum);
}
```

Mr. Mohammed Afzal, Asst. Professor in AIML
Mob: +91-8179700193, Email: mdafzal.mbnr@gmail.com



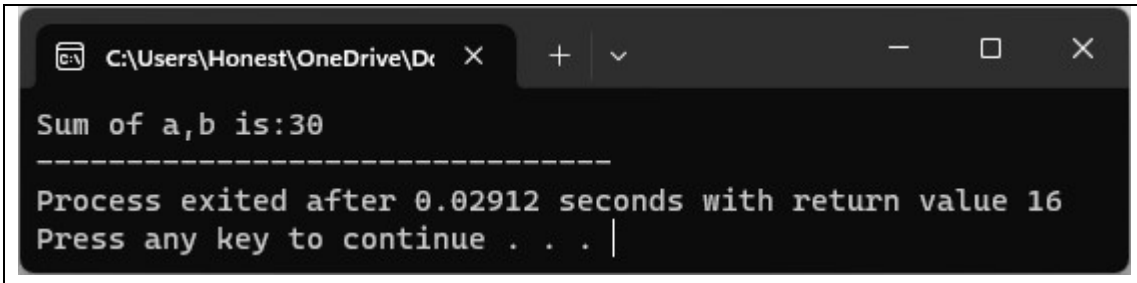
3. Non void functions without parameters

- ✓ This type of functions returns a value but don't have any parameters.
- ✓ The most common use of this design reads data from the keyboard or file and returns to the calling function.
- ✓ As this function returns a specific type of value, the function call must be initialized with variable of that specific type.

// Non - void functions without parameters

nonVoidWOPar.c

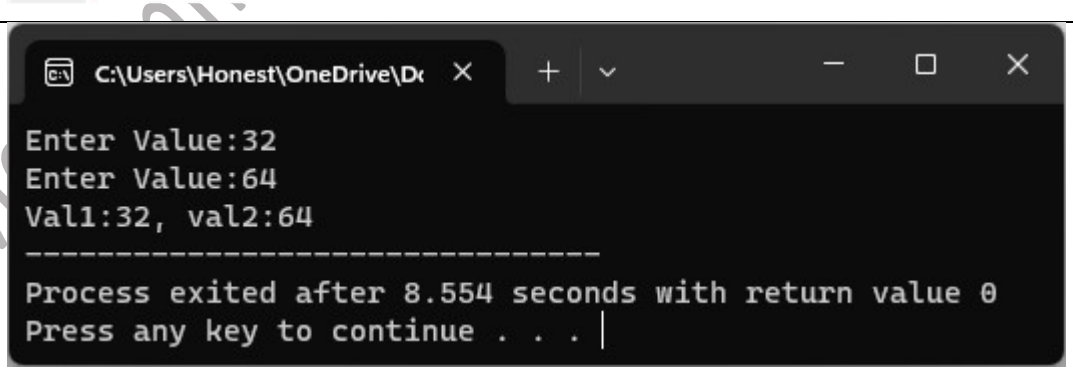
```c
#include <stdio.h>
//Funciton Declaration
int add(void);
int main(void)
{
    int sum;
    //Function Call
    sum=add();
    printf("Sum of a,b is:%d",sum);
    return 0;
}

//Function Definition
int add()
{
    int a=10,b=20,sum;
    sum=a+b;
    return sum;
}
```



//Programs for *getValue()* function

getValue.c

```c
#include <stdio.h>
//Function Declaration
int getValue(void);
int main(void)
{
    int val1,val2;

    //Function Call
    val1=getValue();
    val2=getValue();
    printf("Val1:%d, val2:%d",val1,val2);
    return 0;
}

//Function Definition
int getValue(void)
{
    int val;
    printf("Enter Value:");
    scanf("%d",&val);
    return val;
}
```



4. Non-void functions with parameter

- ✓ This type of functions passes parameters and returns a value.
- ✓ As this function returns a specific type of value, the function call must be initialized with variable of that specific type.

//Non-void function with parameters

nonVoidWPar.c

```c
#include <stdio.h>
//Function Declaration
int add(int,int);
int main(void)
{
    int a=10,b=20,sum;
    //Function Call
    sum=add(a,b);
    printf("Sum of a,b is:%d",sum);
    return 0;
}

//Function Definition
int add(int x,int y)
{
    int res;
    res=x+y;
    return res;
}
```

```
C:\Users\Honest\OneDrive\Dc
Sum of a,b is:30
--------------------------------
Process exited after 0.02486 seconds with return value 0
Press any key to continue . . .
```


User Defined Functions:

- Like every other object in C, functions must be declared, defined and initialized (calling of the function).
- The functions defined by the user who is writing the program, but not by the developers of the language are known as user defined functions.
- To use functions in a C-Program one has to do, the declaration, definition and initialization of the function.

- The function declaration has to be done before function call, which will give the complete picture of the function.
- And definition will succeed function call, which actually contains the statements (body) of the function.

Function Declaration:

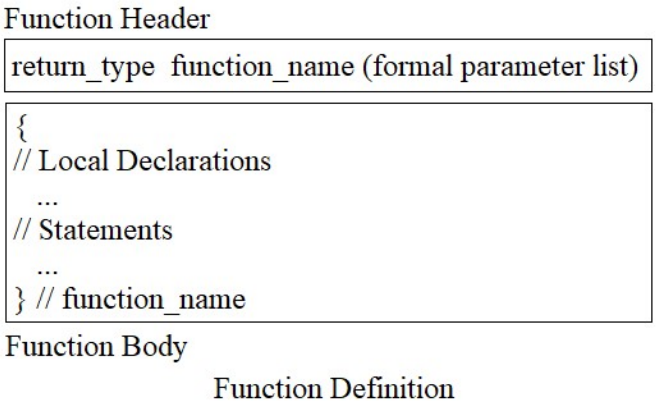
- Function declaration consists of only a function header. It contains no code (body of the function).
- Function declaration consists of three parts: the return type, the function name and the formal parameter list.
- Function declaration will be terminated by a semicolon.
- The return type is the type of value the function is going to return after performing its job. If there is no return type the type will be void type.
- In declaration the parameter list will indicate only the valid type and order of the parameters; the variable identifier can be omitted here in this case.
- Generally, the declarations are placed in the global declaration area of the program. That is before the main function.

Function call:

- The function call is postfix expression.
- The operand in the function call is the function name and the operator is the parenthesis set, which contains the actual parameters.
- The actual parameters hold the actual values that are to be sent to the called function. They must match with the function definition's formal parameter list in both type and order. While specifying actual parameters variable identifiers are required.
- If the function call is having multiple parameters to be passed, they must be separated by commas.
- Functions can return values. If a function's return type is void, then it cannot be used as part of an expression, it has to be called as a stand-alone.
- But if a function is having a return value, then either its call must be initialized with a suitable type variable or it has to be used in an expression. If the function is called like a stand-alone then the return value will be discarded.

Function Definition:

- The function definition contains the code for a function.
- It is made up of two parts: the function header and the function body.



Function Definition

function header

- A function header consists of three parts: the return type, the function name, and the formal parameter list.
- A semicolon is not required at the end of the function definition header.
- The return-type should come before the function name; return-type specifies the type of value a function will return. If the function has nothing to return, the return type will be void.
- The formal parameter list must match with the type and order of the actual parameters. Here while specifying the formal parameters, the variable identifier is must, which will hold the values passed by actual parameters.

function body

- The function body contains the local declarations and the function statements.
- The body starts with local definitions that specify the variables needed by the function. After local definition, the function statements, terminating with a return statement.
- If a function return type is void, it can be written without a return statement.

Formal Parameter List:

- In the definition of a function, the parameters are contained in the formal parameter list.
- This list declares and defines the variables that will contain the data received by the function.
- If the function has no parameters; if it is not receive any data from the calling function, then the parameter list can be void.

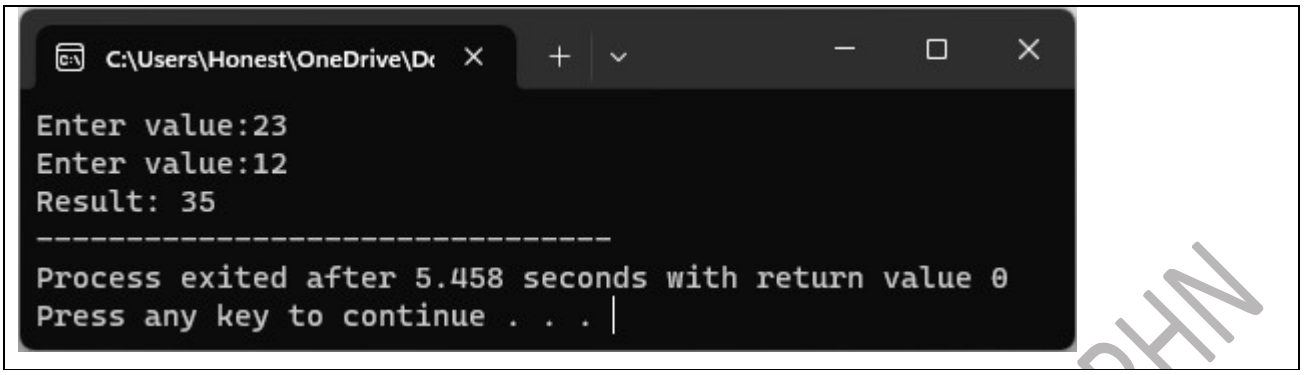
Local Variables:

- A local variable is a variable that is defined inside a function body and used without having any role in the communication between functions.

//Example program for functions

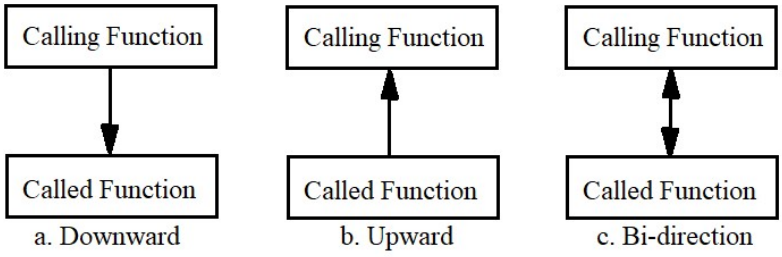
demoFunc.c

```c
#include<stdio.h>
//Function Declaration
int add(int,int);
int getValue();
void putValue(int);

int main(void)
{
    int a,b,c;
    //Function Call
    a=getValue();
    b=getValue();
    c=add(a,b);
    putValue(c);
    return 0;
}

//Function Definition
int add(int x,int y)
{
    int z;
    z=x+y;
    return z;
}

int getValue()
{
    int temp;
    printf("Enter value:");
    scanf("%d",&temp);
    return temp;
}

void putValue(int res)
{
    printf("Result: %d",res);
}
```



INTER-FUNCTION COMMUNICATION:

- Inter function communication is a concept through which calling function and called function will communicate with each other, and exchange the data.
- The data flow between the called and calling function can be divided into three strategies:
  - I. Downward flow
  - II. Upward flow
  - III. Bi-directional flow



Data Flow Strategies

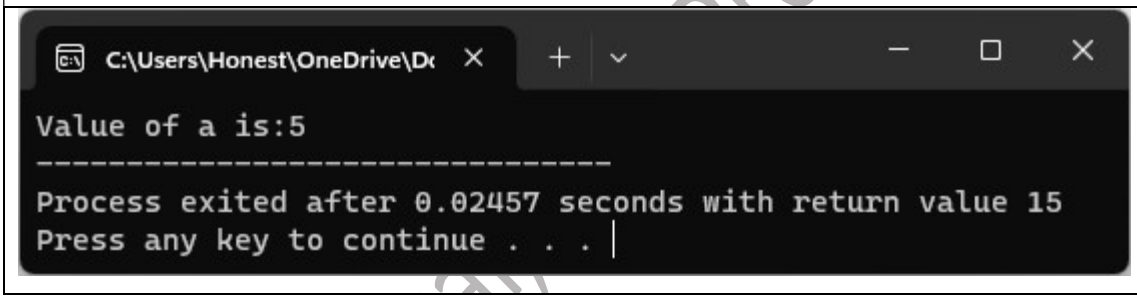
I. Downward flow:

- ✓ In downward communication, the calling function sends data to the called function.
- ✓ No data flows in the opposite directions.
- ✓ The called function may change the values passed, but the original values in the calling function remains untouched.
- ✓ The pass by value or call by value mechanism is a perfect solution for downward flow.
- ✓ A variable is declared and defined in the called function for each value to be received from the calling function.
- ✓ Downward communication is one-way communication. The calling function can send data to the called function, but the called function cannot send any data to the calling function.

//downward communication

downFunc.c

```c
#include<stdio.h>
//Function Declaration
void downFun(int,int);
int main(void)
{
    int a=5;
    //Function Call
    downFun(a,15);
    printf("Value of a is:%d",a);
}

//Function Definition
void downFun(int x,int y)
{
    x=x+y;
}
```



II. Upward Communication (call by reference mechanism):

- ✓ Upward communication occurs when the called function sends data back to the calling function without receiving any data from it.
- ✓ C provides only one way for upward flow, the return statement.
- ✓ Using return statement only one piece of data item can be returned.
- ✓ To send multiple values back to the calling function, we have to use references of variables. References are memory locations which carry the data from called function to the calling function.

//upward communication

upwardFunc.c

```c
#include<stdio.h>
//Function Declaration
void upFun(int*,int*);
void main(void)
{
```

```
    int a,b;
    //Function Call
    upFun(&a,&b);
    printf("Value of a, b are:%d,%d",a,b);
}

//Function Definition
void upFun(int* aptr,int* bptr)
{
    *aptr = 24;
    *bptr = 32;
}
```

```
C:\Users\Honest\OneDrive\Dc
Value of a, b are:24,32
--------------------------------
Process exited after 0.02672 seconds with return value 23
Press any key to continue . . .
```

III. Bi-direction Communication (call by reference mechanism):

- ✓ Bi-directional communication occurs when the calling function sends data down to the called function. After processing the called function sends data up to the calling.
- ✓ The strategy described for upward communication can also be used for bi-direction communication, but with a little modification.
- ✓ The only change is that the indirect reference must be used in both sides of the assignment statement.
- ✓ With this change the variable in the called function first is accessed for retrieving data using address variable in the right-hand side.
- ✓ The same parameter is used again to store a value in the left-hand side.
- ✓ **Note:** both upward and bidirectional communications are examples for pass by reference or call by reference mechanisms.

//Bi-directional communication:

biFunc.c

```
#include<stdio.h>
//Function Declaration
void biFun(int*,int*);
int main(void)
{
    int a=2,b=3;
```

```c
    //Function Call
    biFun(&a,&b);
    printf("Value of a, b are:%d,%d",a,b);
    return 0;
}

//Function Definition
void biFun(int* aptr,int* bptr)
{
    *aptr = *aptr+24;
    *bptr = *bptr+32;
}
```

```
C:\Users\Honest\OneDrive\Dc
Value of a, b are:26,35
--------------------------------
Process exited after 0.0241 seconds with return value 0
Press any key to continue . . .
```

PASSING ARRAY TO FUNCTIONS

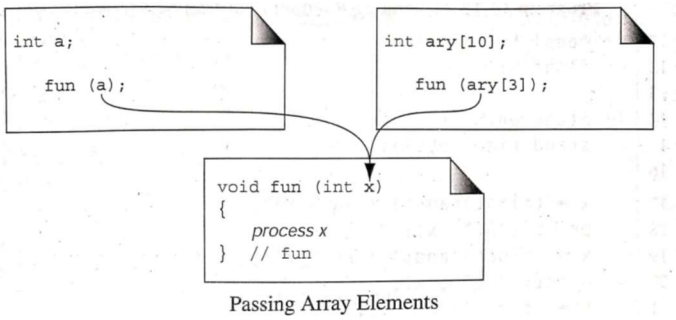
- ✓ Arrays can also be passed as parameters to functions when required.
- ✓ There are two possibilities in which arrays can be passed:
    a) Passing individual elements
    b) Passing the whole array

Passing Individual elements

Individual elements can be either by their values or by their addresses.

Passing data values

- ✓ Individual data values of arrays can be passed using their index along with array name.
- ✓ As long as the type is matching, the called function is unaware whether the value is coming from an array or a normal variable.



Passing Array Elements

Mr. Mohammed Afzal, Asst. Professor in AIML
Mob: +91-8179700193, Email: mdafzal.mbnr@gmail.com

Passing data values from an array

passDataArr.c

```c
#include<stdio.h>
//Function Declaration
void arrAccess(int);
int main(void)
{
    int a[5]={2,4,3,6,9};
    //Function Call
    arrAccess(a[3]);
    printf("\nArray element after function call: %d",a[3]);
}

//Function Definition
void arrAccess(int temp)
{
    printf("Parameter received from calling function: %d",temp);
    temp = 36;
}
```

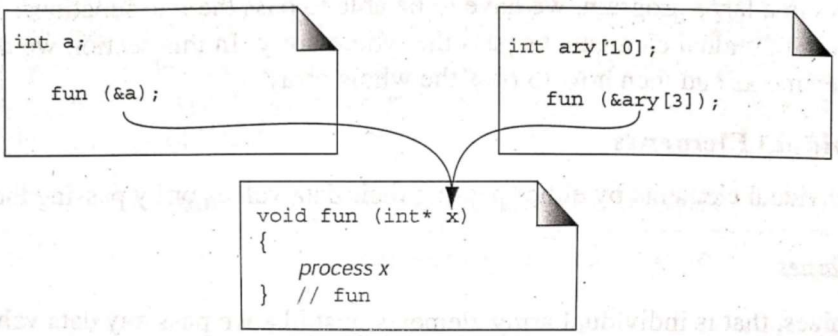
```
C:\Users\Honest\OneDrive\Dc
Parameter received from calling function: 6
Array element after function call: 6
--------------------------------
Process exited after 0.02871 seconds with return value 37
Press any key to continue . . .
```

Passing Addresses

- ✓ The address of the elements of the array can also be passed to the function.
- ✓ The individual element addresses can be passed just like any other variable address.
- ✓ To pass an array element's address, the address operator can be prefixed to the element's indexed reference.
- ✓ Example: The address of 4th element in the array can be passed in following way: &arr[3];

Passing an address of an array element requires two changes in the called function.

- First, it must declare that it is receiving an address.
- Second, it must use the indirection operator (*) to refer to the elements value.



Passing the Address of an Array Element

Passing the address of an array element

passAddrArr.c

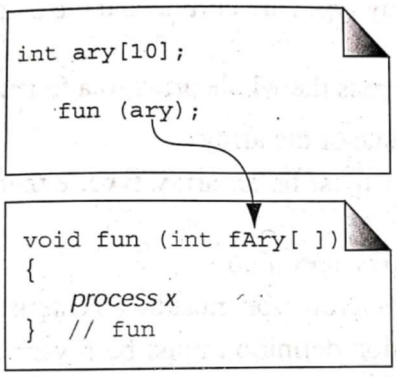
```c
#include<stdio.h>
//Function Declaration
void arrAccess(int*);
void main(void)
{
    int a[5]={2,4,3,6,9};
    //Function Call
    arrAccess(&a[3]);
    printf("\nArray element after function call: %d",a[3]);
}

//Function Definition
void arrAccess(int* temp)
{
    printf("Parameter received from calling function: %d",*temp);
    *temp = 36;
}
```

```
C:\Users\Honest\OneDrive\Dc
Parameter received from calling function: 6
Array element after function call: 36
--------------------------------
Process exited after 0.04537 seconds with return value 38
Press any key to continue . . .
```

Passing the whole array

- ✓ In C, the name of an array holds the address of the first element in the array.
- ✓ Using array name and index values the complete array elements can be accessed.

✓ Passing the array name instead of a single element, allows the called function to refer to the same array back in the calling function.

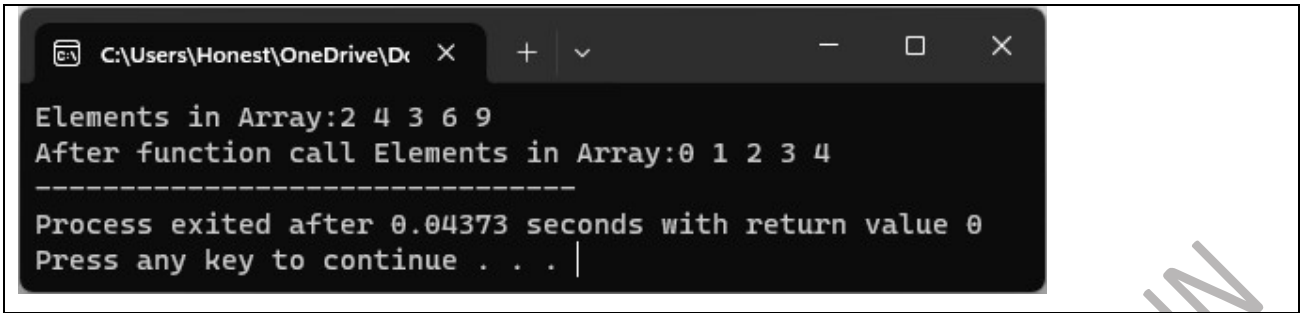


Passing the Whole Array

Passing whole array to called function

passWholeArr.c

```c
#include<stdio.h>
//Function Declaration
void arrAccess(int[]);
int main(void)
{
    int a[5]={2,4,3,6,9},i;
    //Function Call
    arrAccess(a);
    printf("\nAfter function call Elements in Array:");
    for(i=0;i<5;i++)
        printf("%d ", a[i]);
    return 0;
}

//Function Definition
void arrAccess(int arrTemp[])
{
    int i;
    printf("Elements in Array:");
    for(i=0;i<5;i++)
        printf("%d ", arrTemp[i]);

    for(i=0;i<5;i++)
        arrTemp[i] = i;
}
```



STANDARD FUNCTIONS (LIBRARY FUNCTIONS):

- ✓ C provides a rich collection of standard functions whose definitions have been written and are ready to be used in our programs.
- ✓ To use these functions, we must include their function declarations into our program.
- ✓ The function declarations are generally found in header files.
- ✓ Instead of adding each declaration separately, a simple statement with header file can be included on top of the program.

Math functions

- ✓ Several library functions are available for mathematical calculations.
- ✓ Most of them are in either math header file (math.h) or standard library (stdlib.h).

Absolute value functions:

- ✓ These functions will return the absolute value of a number.
- ✓ An absolute value is the positive value of the value regardless of its sign.
- ✓ There are 3-integer and 3-floating point functions.
- ✓ The integer functions are *abs*, *labs*, and *llabs*.
- ✓ For *abs* the parameter must be an *int* and it returns an *int*. For *labs* the parameter must be a *long int*, and it returns a *long int*. For *llabs* the parameter must be a *lond int*, and it returns a *long long int*.

General Syntax:
int abs(int); *long labs(long);* *long long llabs(long long);*

- ✓ The real functions are *fabs*, *fabsf* and *fabsl*. For fabs the parameter is a *double*, and it returns a *double*. For *fabsf* the parameter is a *float* and returns a *float*. For *fabsl* the parameter is a *long double* and returns a *long double*.

General Syntax:
double fabs(double);
float fabsf(float);
long double fabsl(long double);

Absolute value functions

absFun.c

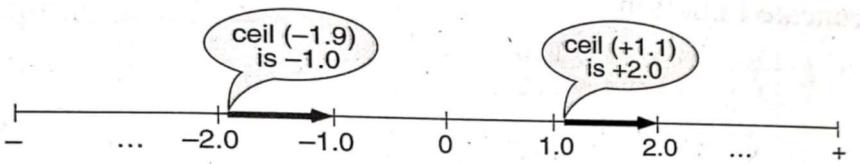
```c
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val=-2.674;
    float res;
    res = fabsf(val);
    printf("Absolute value of %f is %f", val, res);
}
```

```
C:\Users\Honest\OneDrive\Dc
Absolute value of -2.674000 is 2.674000
--------------------------------
Process exited after 0.2929 seconds with return value 39
Press any key to continue . . .
```

Ceiling Functions

- ✓ A ceiling is the smallest integral value greater than or equal to a number.

  Example: Ceiling of 3.00001 is 4.
- ✓ If the complete numbers are considered as continues range from minus infinity to plus infinity, this function moves the number right towards an integral value.



Ceiling Function

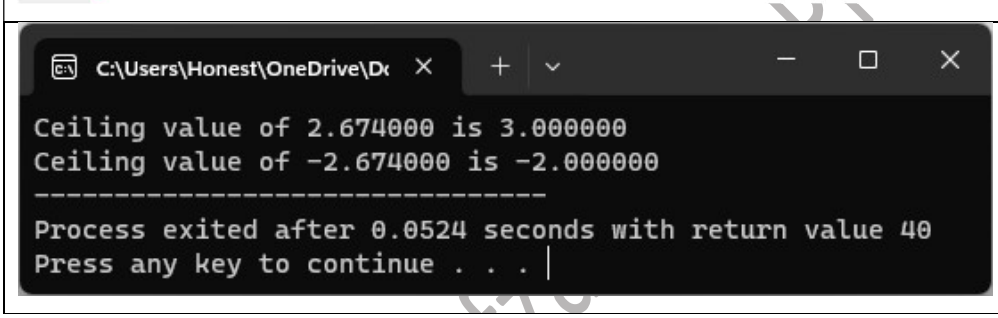
The declarations for ceiling function are:
double ceil (double);
float ceilf(float);
long double ceill(long double);

Example:
ceil (-1.9) returns -1.0 *ceil (1.1) returns 2.0*

Ceiling Functions

ceilFun.c

```c
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val = 2.674;
    float res;
    res = ceilf(val);
    printf("Ceiling value of %f is %f", val, res);
    val = -2.674;
    res = ceilf(val);
    printf("\nCeiling value of %f is %f", val, res);
}
```



Floor Functions

- ✓ A floor is the largest integral value that is equal to or less than a number.
- ✓ **Example:** Floor of 3.99999 is 3.0.
- ✓ On number series, this function moves the number left towards an integral value.

The declarations for floor function are:
double floor (double); *float floorf (float);* *long double floorl (long double);*
Example:
floor (-1.1) returns -2.0 *floor (1.9) returns 1.0*

Floor Functions

floorFun.c

```c
#include<stdio.h>
#include<math.h>
int main(void)
{
    float val = 2.674;
    float res;
    res = floorf(val);
    printf("Floor value of %f is %f", val, res);
    val = -2.674;
    res = floorf(val);
    printf("\nFloor value of %f is %f", val, res);
    return 0;
}
```

```
C:\Users\Honest\OneDrive\Dc
Floor value of 2.674000 is 2.000000
Floor value of -2.674000 is -3.000000
--------------------------------
Process exited after 0.02811 seconds with return value 0
Press any key to continue . . .
```

Truncate Functions

- ✓ The truncate functions return the integral value in the direction of 0.
- ✓ They are same as floor functions for positive numbers and same as ceiling function for negative numbers.

Their function declarations are:
double trunc(double); *float truncf(float);* *long double truncl(long double);*
Example:
trunc (-1.1) returns -1.0 *trunc (1.9) returns 1.0*

Truncate Functions

truncFun.c

```c
#include<stdio.h>
#include<math.h>
int main(void)
{
    float val = 2.674;
    float res;
    res = truncf(val);
    printf("Truncate value of %f is %f", val, res);
    val = -2.674;
    res = truncf(val);
    printf("\nTruncate value of %f is %f", val, res);
    return 0;
}
```

```
C:\Users\Honest\OneDrive\Dc
Truncate value of 2.674000 is 2.000000
Truncate value of -2.674000 is -2.000000
--------------------------------
Process exited after 0.02933 seconds with return value 0
Press any key to continue . . .
```

Round Functions

- ✓ The round functions return the nearest integral value.

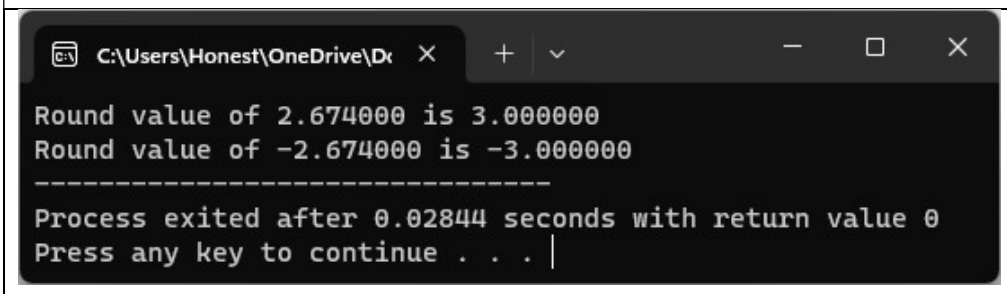
Their function declarations are:
double round(double);
float roundf(float);
long double roundl(long double);
Example:
round(-1.1) returns -1.0
round(1.9) returns 2.0

Round Functions

roundFun.c

```c
#include<stdio.h>
#include<math.h>
int main(void)
{
    float val = 2.674;
    float res;
    res = roundf(val);
    printf("Round value of %f is %f", val, res);
    val = -2.674;
```

```
    res = roundf(val);
    printf("\nRound value of %f is %f", val, res);
    return 0;
}
```



Power Functions:

- ✓ The power (pow) function returns the value of the x raised to the power of y.
- ✓ An error occurs if the base (x) is negative and the exponent (y) is not an integer, or if the base is zero and the exponent is not positive.

The power function declarations are:
double pow(double, double); *float powf(float, float);* *long double powlf(long double, long double);*
Example:
pow (3.0, 4.0) returns 81.0

Square Root Functions:

- ✓ The square root functions return the non-negative square root of a number.
- ✓ An error occurs if the number is negative.

The square root function declarations are:
double sqrt(double); *float sqrtf(float);* *long double sqrtlf(long double);*
Example:
sqrt (25) returns 5.0

Power and Square root functions

powSqrt.c

```c
#include<stdio.h>
#include<math.h>
void main(void)
{
    float val1 = 3;
    float val2 = 4;
    float res1, res2;
    res1 = powf(val1,val2);
    printf("%f power of %f is %f", val1, val2, res1);
    res2 = sqrtf(res1);
    printf("\nSquare root of %f is %f", res1, res2);
}
```

```
C:\Users\Honest\OneDrive\Dc
3.000000 power of 4.000000 is 81.000000
Square root of 81.000000 is 9.000000
--------------------------------
Process exited after 0.03984 seconds with return value 37
Press any key to continue . . .
```


RECURSION

- ✓ Programmers use two approaches for writing repetitive algorithms: One approach uses loops and the other uses recursion.
- ✓ Recursion is a repetitive process in which a function calls itself.
- ✓ Some older languages do not support recursion. One major language that does not support recursion is COBOL.
- ✓ To study the concept of recursion let us consider the example of finding factorial of a given number.
- ✓ In our study we'll see both iterative definition and recursive definition for finding the factorial.

Factorial of a Number

Iterative Definition

- ✓ The factorial of a number is the product of the integral values from 1 to the given number.
- ✓ The iterative definition can be shown as:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) \ldots 3 * 2 * 1 & \text{if } n > 0 \end{cases}$$

Iterative Factorial Definition

Mr. Mohammed Afzal, Asst. Professor in AIML
Mob: +91-8179700193, Email: mdafzal.mbnr@gmail.com

- ✓ An iterative function can be defined which will take only the iteration counter to calculate the factorial but not the function call.

  Example: factorial (4) = 4 * 3 * 2 * 1 = 24

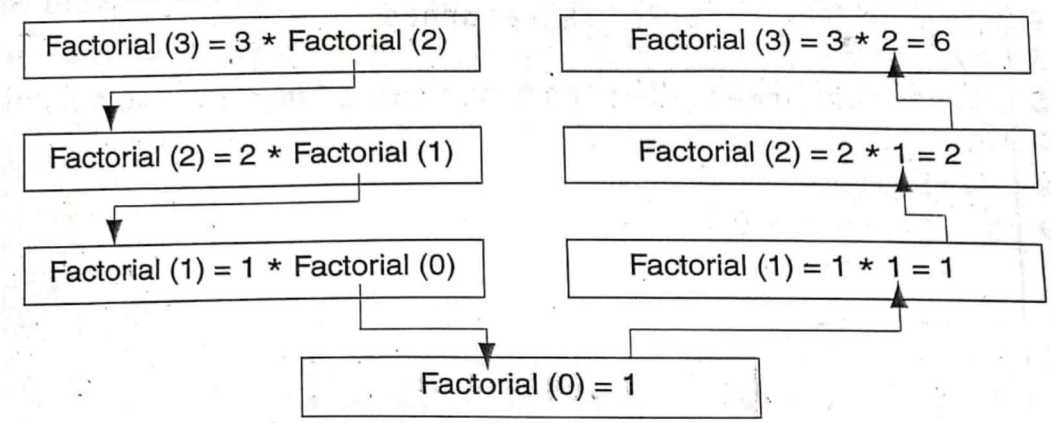
Recursive Definition

- ✓ A function is defined recursively whenever the function itself appears within the definition itself.
- ✓ The recursive definition of factorial function can be shown as:

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

Recursive Factorial Definition

- ✓ The decomposition of factorial (3), using the above formula is shown in the following formula.



Factorial (3) Recursively

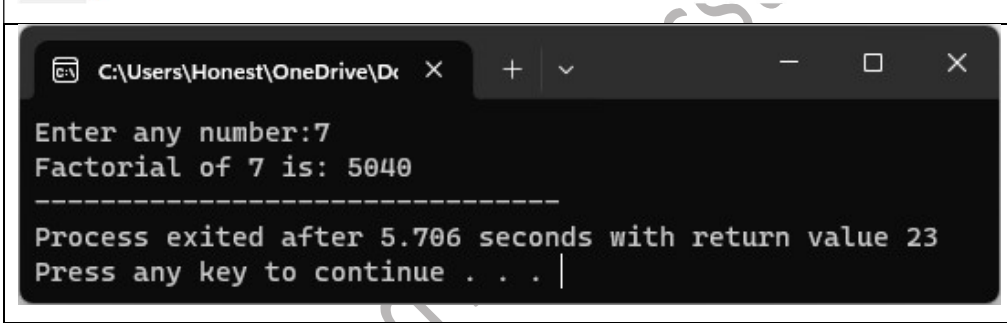
- ✓ Observe it carefully; the recursive solution for a problem involves a two-way journey: the decomposition will be from top to bottom, and then we solve it from bottom to the top.
- ✓ Every recursive call must either solve part of the problem or reduce the size of the problem.
- ✓ The recursive calculation looks more difficult when using pen and paper, but it is often much easier and more elegant solution when a computer is used.
- ✓ Every recursive solution to the problem will have a base-case and general-case. A base-case will provide the solution to the problem and all other cases will be known as general-cases.
- ✓ In the factorial example the base-case is factorial of (0) and all other cases are general-cases.

Iterative solution to factorial problem

Iterative solution to factorial problem

iterFact.c

```c
#include<stdio.h>
//Function Declaration
int iteraFact(int);
void main(void)
{
    int n,res;
    printf("Enter any number:");
    scanf("%d",&n);
    //Fucntion Call
    res=iteraFact(n);
    printf("Factorial of %d is: %d", n, res);
}

//Function Definition
int iteraFact(int num)
{
    int i, fact = 1;
    for(i=1;i<=num;i++)
        fact=fact*i;
    return fact;
}
```



Recursive solution to factorial problem

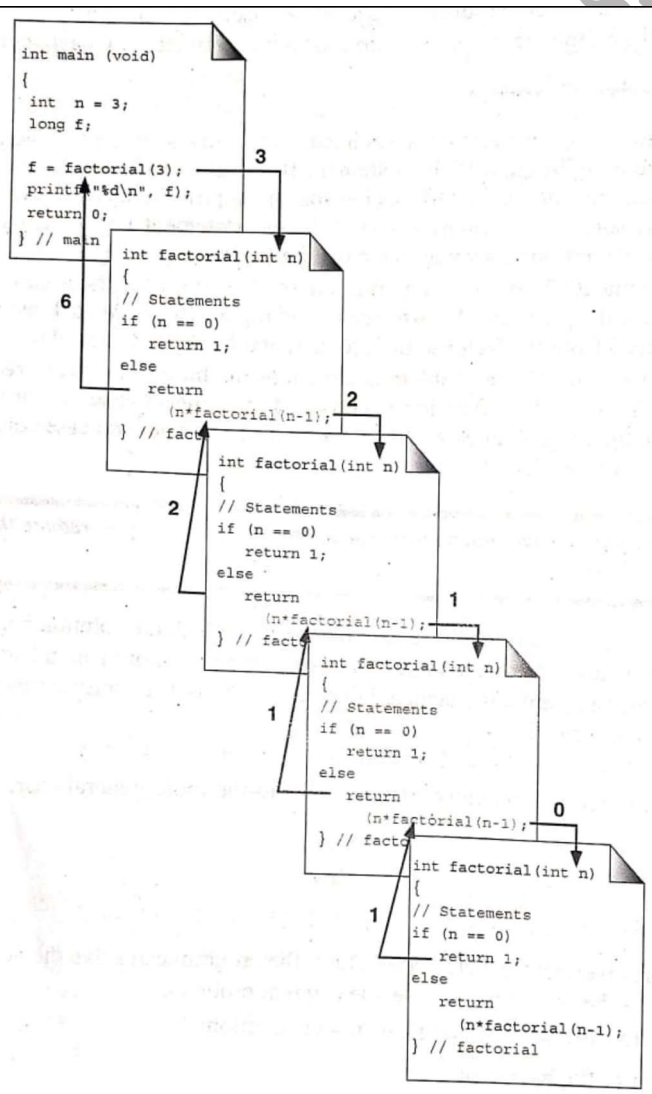
Recursive solution to factorial problem

recurFact.c

```c
#include<stdio.h>
//Function Declaration
int recurFact(int);
void main(void)
{
    int n,res;
    printf("Enter any number:");
    scanf("%d",&n);
    //Function Call
    res=recurFact(n);
    printf("Factorial of %d is: %d", n, res);
}
```

```
13
14  //Function Definition
15  int recurFact(int num)
16  {
17      if(num == 0)
18          return 1;
19      else
20          return (num * recurFact(num-1));
21  }
```

```
C:\Users\Honest\OneDrive\Dc
Enter any number:8
Factorial of 8 is: 40320
--------------------------------
Process exited after 4.147 seconds with return value 24
Press any key to continue . . .
```

Calling a Recursive Function

Mr. Mohammed Afzal, Asst. Professor in AIML
Mob: +91-8179700193, Email: mdafzal.mbnr@gmail.com

Fibonacci Numbers

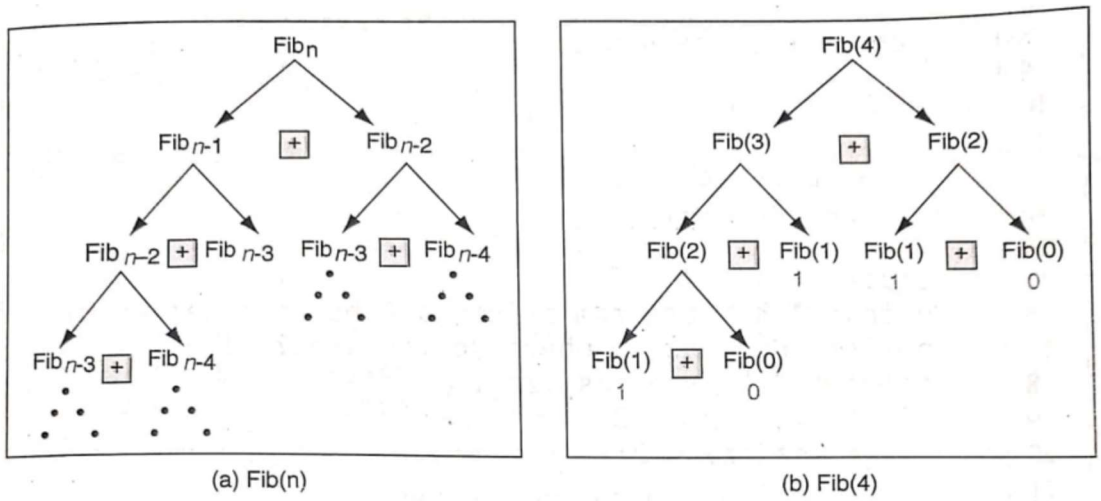
- ✓ Finding the Fibonacci numbers is another example for recursion.
- ✓ Fibonacci numbers are a series in which each number is the sum of the previous two numbers. This number series is named after an Italian mathematician Leonardo Fibonacci of thirteenth century.
- ✓ **Example:** First few numbers in Fibonacci series are:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ….
- ✓ To start the series always first two numbers will be 0 and 1.
- ✓ The generalized statements for Fibonacci series are as follows:

  Given: $fibonacci_0 = 0;$

  $fibonacci_1 = 1;$

  Then: $fibonacci_n = fibonacci_{n-1} + fibonacci_{n-2}$
- ✓ To calculate the fibonacci4 the steps are shown in the following figure:



Fibonacci Numbers

Iterative solution to Fibonacci Series problem

Iterative solution to Fibonacci Series Problem

iterFib.c

```c
#include <stdio.h>
//Function Declaration
void iterFibo(int);
void main(void)
{
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    //Function Call
    iterFibo(n);
}
```

```c

//Function Definition
void iterFibo(int n)
{
    int i, term1 = 0, term2 =1, nextTerm;
    for (i = 1; i <= n; ++i)
    {
        printf("%d, ", term1);
        nextTerm = term1 + term2;
        term1 = term2;
        term2 = nextTerm;
    }
}
```

```
C:\Users\Honest\OneDrive\D
Enter the number of terms: 8
0, 1, 1, 2, 3, 5, 8, 13,
--------------------------------
Process exited after 2.45 seconds with return value 9
Press any key to continue . . .
```


Recursive solution to Fibonacci Series problem

Recursive solution to Fibonacci Series problem

recurFib.c

```c
#include <stdio.h>
//Function Declaration
int recurFib(int);
int main(void)
{
    int nterms, fib = 0, i;
    printf("Enter the nuumber of terms:");
    scanf("%d", &nterms);
    printf("Fibonacci series terms are:\n");
    for (i = 0; i < nterms; i++)
    {
        printf("%d, ", recurFib(fib));
        fib++;
    }
    return 0;
}
```

```c

//Function Definition
int recurFib(int n)
{
    if(n == 0 || n == 1)
        return n;
    return (recurFib(n - 1) + recurFib(n - 2));
}
```

```
C:\Users\Honest\OneDrive\Dc
Enter the nuumber of terms:8
Fibonacci series terms are:
0, 1, 1, 2, 3, 5, 8, 13,
--------------------------------
Process exited after 2.878 seconds with return value 0
Press any key to continue . . .
```

Finding GCD (Greatest Common Divisor):

- ✓ The HCF (Highest Common Factor) or GCD (Greatest Common Divisor) of two integers is the largest integer that can exactly divide both numbers (without a remainder).

Iterative solution to GCD problem

Iterative solution to GCD problem

iterGCD.c

```c
#include <stdio.h>
void main(void)
{
    int n1, n2, i, gcd;
    printf("Enter two integers: ");
    scanf("%d %d", &n1, &n2);
    for(i=1; i <= n1 && i <= n2; i++)
    {
        // Checks if i is factor of both integers
        if(n1%i==0 && n2%i==0)
            gcd = i;
    }
    printf("G.C.D of %d and %d is %d", n1, n2, gcd);
}
```

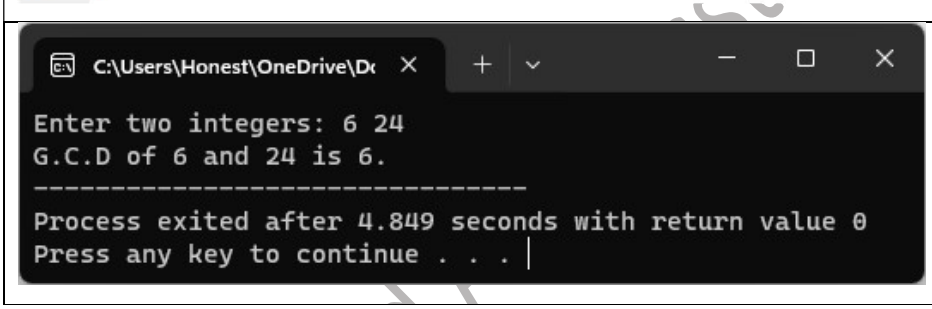
```
C:\Users\Honest\OneDrive\Dc
Enter two integers: 12 24
G.C.D of 12 and 24 is 12
--------------------------------
Process exited after 6.928 seconds with return value 24
Press any key to continue . . .
```

Recursive solution to GCD problem

Recursive solution to GCD problem

recurGCD.c

```c
#include <stdio.h>
//Function Declaration
int gcd(int, int);
int main(void)
{
    int n1, n2;
    printf("Enter two integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2, gcd(n1,n2));
    return 0;
}

//Function Definition
int gcd(int n1, int n2)
{
    if (n2 != 0)
        return gcd(n2, n1%n2);
    else
        return n1;
}
```



MEMORY ALLOCATION FUNCTIONS:

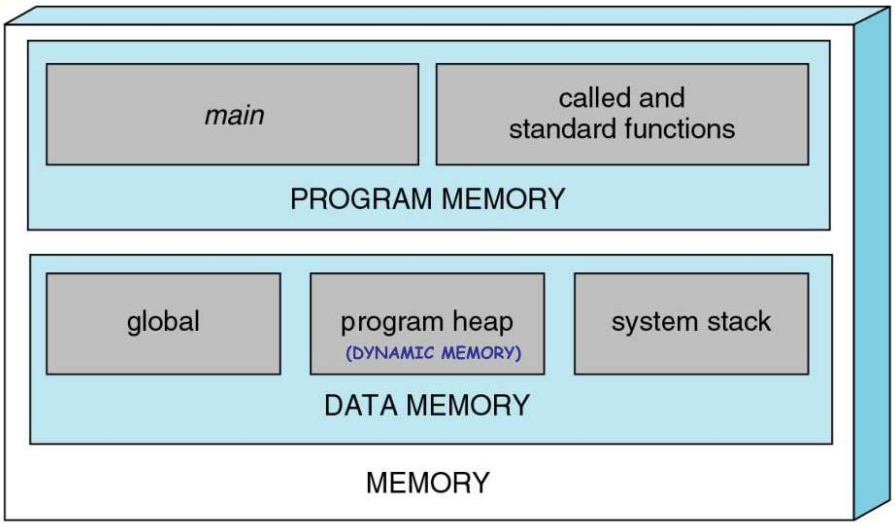
In C, there are two ways to reserve memory locations for variables.

1. Static memory allocation
2. Dynamic memory allocation

Memory Usage:

- ✓ Memory allocated to a program is divided into program memory and data memory.
- ✓ Program memory consists of memory used for main and all called functions.
- ✓ Data memory consists of permanent definitions of global and local data and constants, dynamic data memory.

- ✓ main() function must be in memory all the time. Each called function must be in memory only when it is called and becomes active.
- ✓ If a function called more than once, the copies of the local variables of this function will be maintained inside stack memory.
- ✓ In addition to the stack memory, a separate memory area known as heap is also available. Heap memory is unused memory allocated to the program and available to be assigned during execution. It is the memory pool from which the memory is allocated when requested by the memory allocation functions.



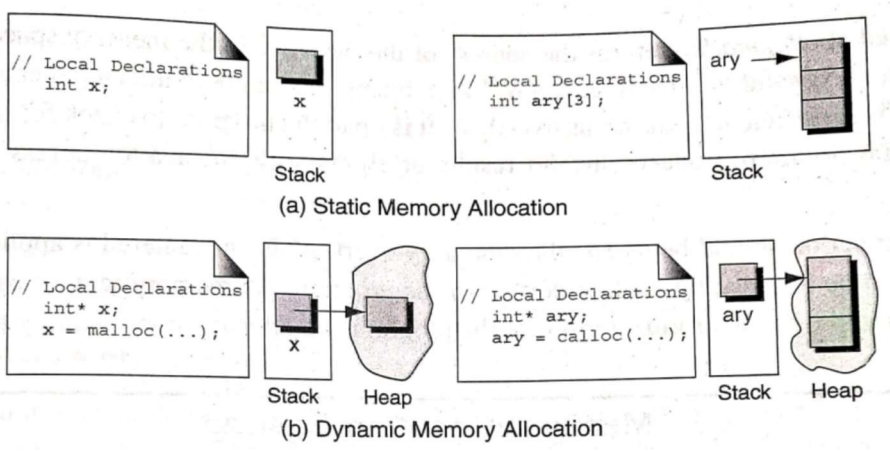
A Conceptual View of Memory

Static Memory Allocation:

- ✓ Static memory allocation requires that the declaration and definition of memory be fully specified in the source program.
- ✓ The number of bytes reserved cannot be changed during run time of the program.
- ✓ So far, we used this technique for reserving memory for variables, arrays and pointers.

Dynamic Memory Allocation:

- ✓ Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running.
- ✓ Unlike static memory allocations, dynamic memory allocations have no identifier associated with it. It can be accessed only through an address. That is to access data in dynamic memory we need a pointer.



Memory Allocation Functions

- ✓ Four memory management functions are used with dynamic memory.
- ✓ Three of them *malloc(), calloc()* and *realloc()* are used to allocate the memory.
- ✓ The fourth one free is used to return the memory when it is no longer needed.
- ✓ All these memory management functions are found in the standard library file *(stdlib.h)*.

Block Memory allocation (malloc())

- ✓ The *malloc()* function allocates a block of memory that contains the number of bytes specified in its parameter.
- ✓ The allocated memory is not initialized and contains unknown values.

  Syntax: *void* malloc(int size);*
- ✓ On successful allocation it returns a void pointer to the first byte of the allocated memory. If the allocation of memory is unsuccessful the function will return a NULL pointer.
- ✓ Calling a malloc function with a zero size is a problem, the results are unpredictable. It may return a NULL pointer or it may return some other value. Never call a malloc with a zero size.

Demonstration of malloc function

memAlloc.c

```c
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
    int* ptr;
    int n, i, sum = 0;
    printf("Enter the number of elements you need from heap:");
    scanf("%d",&n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));
    // Check if the memory has been successfully allocated by malloc or not
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
```

```c
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using malloc.\n");
        printf("Enter %d elements into the heap:",n);
        for (i = 0; i < n; i++)
            scanf("%d",&ptr[i]);

        // Print the elements from the heap
        printf("The elements in heap are: ");
        for (i = 0; i < n; ++i)
            printf("%d ", ptr[i]);
    }
    free(ptr);
}
```

```
C:\Users\Honest\OneDrive\Dc
Enter the number of elements you need from heap:6
Memory successfully allocated using malloc.
Enter 6 elements into the heap:2 3 5 9 8 7
The elements in heap are: 2 3 5 9 8 7
--------------------------------
Process exited after 10.54 seconds with return value 1
Press any key to continue . . .
```

Contiguous Memory allocation (calloc())

- ✓ The *calloc()* function is primarily used to allocate memory for arrays.
- ✓ It differs from malloc only in that; it sets memory to null characters.

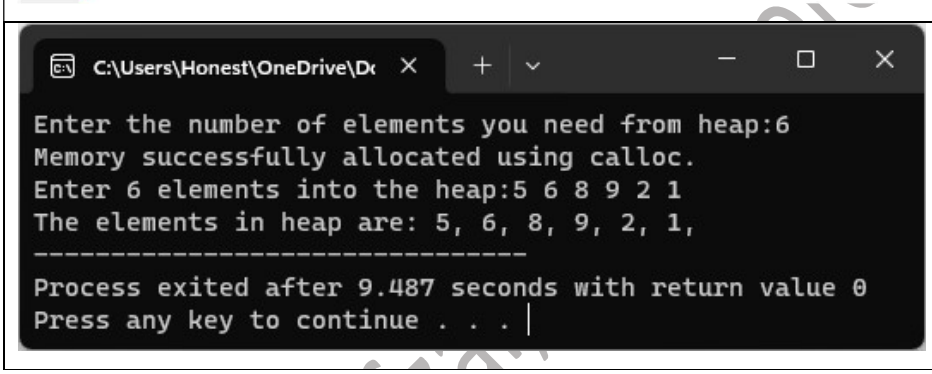
  Syntax: *void* calloc(int element_count, int element_size);*
- ✓ On successful allocation it returns a void pointer to the first byte of the allocated memory. If the allocation of memory is unsuccessful the function will return a *NULL* pointer.

Demonstration of calloc() function

callocMem.c

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int* ptr;
    int n, i, sum = 0;
    printf("Enter the number of elements you need from heap:");
    scanf("%d",&n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

```

```c
    // Check if the memory has been successfully allocated by calloc or not
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using calloc.\n");
        printf("Enter %d elements into the heap:",n);
        for (i = 0; i < n; i++)
            scanf("%d",&ptr[i]);

        // Print the elements from the heap
        printf("The elements in heap are: ");
        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    free(ptr);
    return 0;
}
```



Reallocation of memory (realloc())

- ✓ The *realloc()* function is advised to be used with much care. As a minor issue in the usage will lead to the crash of the program.
- ✓ Given a pointer to a previously allocated block of memory, *realloc()* changes the size of the block by deleting or extending the memory at the end of the block.
- ✓ If the memory cannot be extended because of other allocations, *realloc()* allocates a completely new block, copies the existing memory locations to the new allocations, and deletes the old locations.

Syntax: *void* realloc(void* ptr, int newSize);*

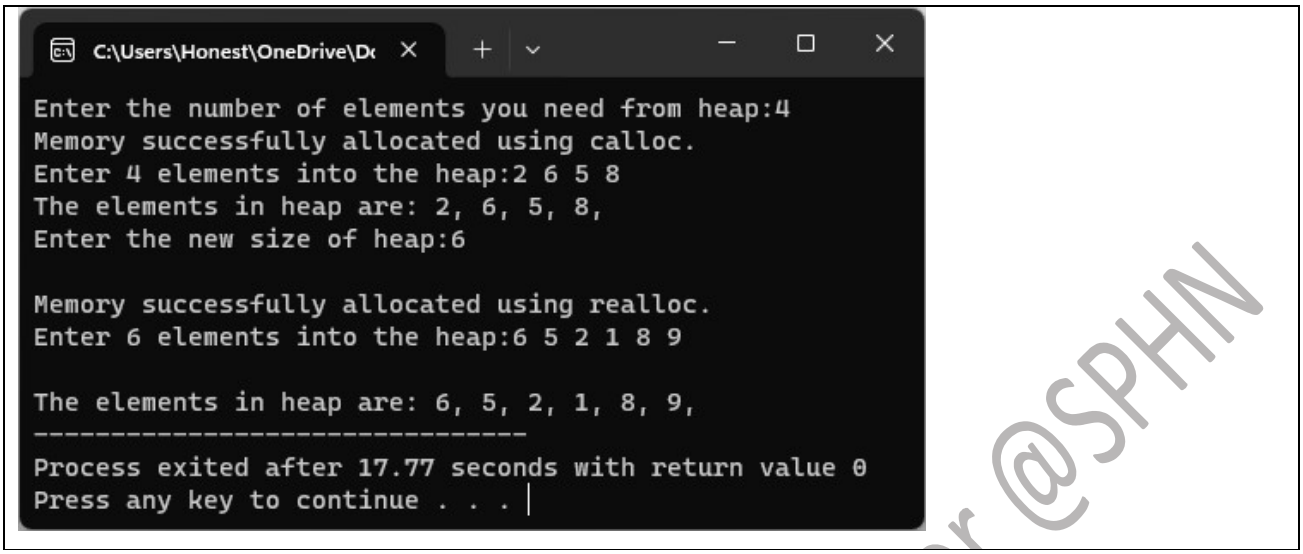
Demonstration of realloc() function

reAlloc.c

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
```

```c
    int* ptr;
    int n, i, sum = 0;
    printf("Enter the number of elements you need from heap:");
    scanf("%d",&n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully allocated by calloc or not
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using calloc.\n");
        printf("Enter %d elements into the heap:",n);
        for (i = 0; i < n; i++)
            scanf("%d",&ptr[i]);

        // Print the elements from the heap
        printf("The elements in heap are: ");
        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    printf("\nEnter the new size of heap:");
    scanf("%d",&n);

    // Dynamically allocate memory using realloc()
    ptr = (int*)realloc((void*)ptr,n * sizeof(int));

    // Check if the memory has been successfully allocated by realloc or not
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
        printf("\nMemory successfully allocated using realloc.\n");
        printf("Enter %d elements into the heap:",n);
        for (i = 0; i < n; i++)
            scanf("%d",&ptr[i]);

        // Print the elements from the heap
        printf("\nThe elements in heap are: ");
        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    free(ptr);
    return 0;
}
```

Mr. Mohammed Afzal, Asst. Professor in AIML
Mob: +91-8179700193, Email: mdafzal.mbnr@gmail.com



Releasing Memory (free())

- ✓ When memory locations allocated by *malloc()*, *calloc()* or *realloc()* are no longer needed, they should be freed using the *free()* function.
- ✓ It is an error to free memory with a *NULL* pointer, a pointer to other than the first element of an allocated block, a pointer that is different type than the pointer that allocated the memory.
- ✓ It is also an error to refer to memory after it has been released.

  Syntax: *void free(void* ptr);*