

A close-up, slightly blurred photograph of a laptop. The screen shows a code editor with syntax-highlighted code, likely for a web application. The code includes HTML tags like <Link>, <div>, <Preview>, <Editor>, and <button>, along with JavaScript-like expressions such as {code} and evalInContext. The keyboard is visible in the foreground, with keys like 'P', 'R', 'T', 'F', 'G', 'C', 'V', 'B', 'N', 'M' clearly visible. A dark semi-transparent overlay covers the lower half of the image, containing the text 'AMIT LEARNING' and 'SESSION 3 FUNCTION AND MODULAR PROGRAMMING'. The 'AMIT' logo is in the bottom right corner.

AMIT LEARNING

SESSION 3 FUNCTION AND MODULAR PROGRAMMING

CONTENTS

1	WHAT IS FUNCTION	1
	1.1 Type of function in c programming	1
	1.2 How user defined function	1
	1.3 Type of user defined function	1
	1.4 Advantages of user defined function .	1
	1.5 Recursive Function	1
2	Scope and life time of variable	2
	2.1 Scope Rules in c	2
	2.2 Automatic , External, Static , Register	2
3	Modular programming	3

➤ **What is function ?**

- The function is one of the most basic things to understand in C programming.
- A function is a sub-unit of a program which performs a specific task.
- Functions may take arguments (variables) and may return an argument.
- A function is a set of statements that take inputs, do some specific computation and produces output.
- The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

➤ **Why do we need functions?**

- Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code modular. Consider a big file having many lines of codes. It becomes really simple to read and use the code if the code is divided into functions.
- Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

1 Type of function in c programming

1.1 Type of function in c programming

- There are two types of function in C programming:
 - Standard library functions.
 - User defined function.

- Standard library functions:
 - The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

 - These functions are defined in the header file. When you include the header file, these functions are available for use
 - ❑ For example:
The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

1 Type of function in c programming

1.1 Type of function in c programming

➤ User-defined function :

- As mentioned earlier, C allow programmers to define functions. Such functions created by the user are called user-defined functions.
- You can create as many user-defined functions as you want.

1 How user defined function

1.2 How user defined function

➤ How user defined function:

1. Function declaration
2. Calling function.
 - Type of calling function:
 - ✓ Call by value.
 - ✓ Call by reference .
3. Function definition.

➤ **Function Declaration:**

A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration. If we ignore function prototype, program may compile with warning, and may work properly. But some times, it will give strange output and it is very hard to find such programming mistakes.

- ✓ Syntax of function prototype:

```
returnType functionName(type1 argument1, type2 argument2,...);
```

➤ **function call:**

- is used to execute the function, and is run from within the main function.
- The OS call main function.

```
functionName(argument1, argument2, ...);
```

- ✓ Syntax of function call

1 How user defined function

1.2 How user defined function

➤ How user defined function:

➤ **function definition :**

is the actual function. The first line of the function definition is identical to the function prototype i.e. return type, the function name and any parameters. Additionally the function definition contains all the code that the function will execute, i.e. statements etc. The function definition unlike the function prototype does not end with a semicolon, but uses curly brackets or braces (`{ }`), these surround the code that forms the function body. The function definition can be placed before the main function, but it is often good practice to place it after.

✓ Syntax of function definition :

```
returnType functionName(type1 argument1, type2 argument2, ...)  
{  
    //body of the function  
}
```

1

How user defined function

1.2 How user defined function

- Type of calling function:
 - ✓ Call by value.
 - ✓ Call by reference
- **Parameter Passing to functions when calling function:**
 - **Actual Parameter :** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.
 - **Formal Parameter :** A variable and its type as they appear in the prototype of the function or method.
- There are two most popular ways to pass parameters.
 - **Pass by Value:** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller. Parameters are always passed by value in C.
 - **Pass by Reference** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

1 How user defined function

1.2 How user defined function

➤ **When to use call by value:**

- When you don't want to change the value of the actual parameters.
- To avoid any side effects.
- Just only want the value of actual parameters.
- Only want to check any condition.
- You don't have memory concerns. Because in the call by value you also need extra memory for formal parameters.

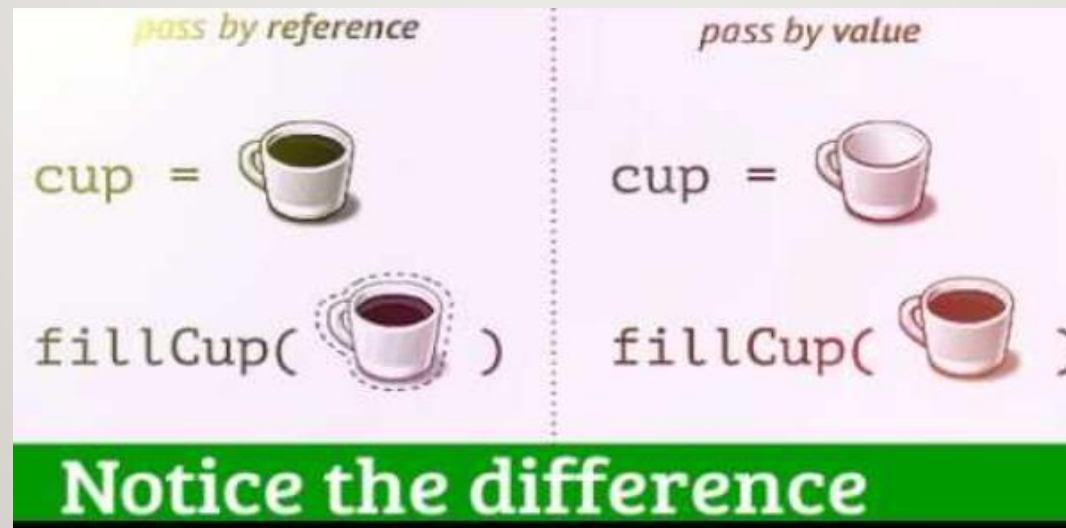
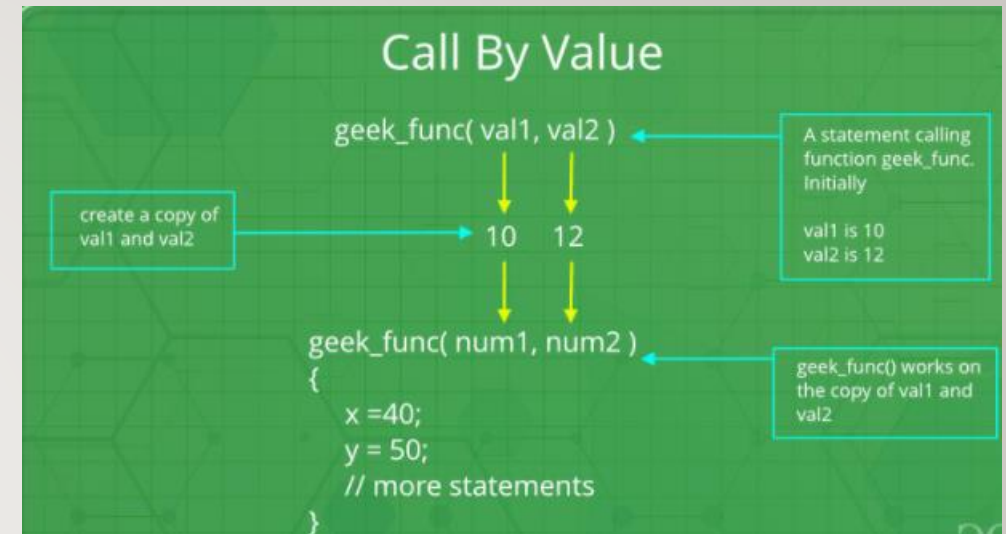
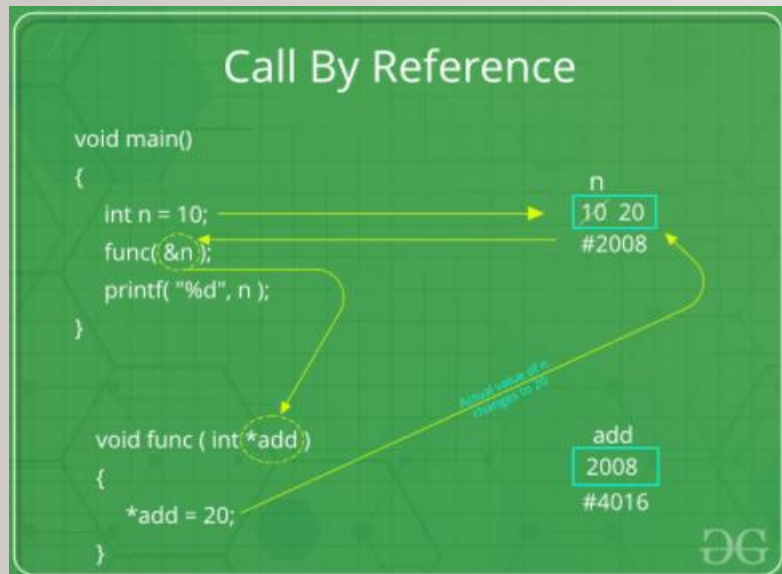
➤ **When to use call by reference:**

- When want to change the value of actual parameters.
- When you have memory concerns then use call by reference. It does not create duplicate memory.
- When you need to pass the structure, union, or big size of data. Because in which we are using the pointer so it helps to reduce the stack usage.
- If you don't want to change the value of structure and union or any data then you can use const with pointers.

1

How user defined function

1.2 How user defined function



1

How user defined function

1.2 How user defined function

CODE

```
// C program to illustrate
// call by value
#include <stdio.h>

void func(int a, int b)
{
    a += b;
    printf("In func, a = %d b = %d\n", a, b);
}

int main(void)
{
    int x = 5, y = 7;

    // Passing parameters
    func(x, y);
    printf("In main, x = %d y = %d\n", x, y);
    return 0;
}
```

OUTPUT

```
In func, a = 12 b = 7
In main, x = 5 y = 7
```

DESCRIPTION

- In this example call by value actual parameter x ,y not change in formal parameter a,b.

1

How user defined function

1.2 How user defined function

CODE

```
void swapnum(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}

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

    // passing parameters
    swapnum(&a, &b);

    printf("a is %d and b is %d\n", a, b);
    return 0;
}
```

OUTPUT

a is 20 and b is 10

DESCRIPTION

- In this example call by reference actual parameter a ,b change in formal parameter i,j.

1 How user defined function

1.2 How user defined function

➤ How user defined function:

```
int a = 10, b = 5, c;
```

```
int product(int x, int y);
```

Function Prototype

- int is the return type and int x and int y are the function arguments

```
int main(void)
```

```
{  
    c = product(a,b);
```

Main Function

- int is always the return type and there are no arguments, hence the (void). Curly braces { } mark the start and end of the main function

```
    printf("%i\n",c);
```

Function call

- product(a,b); a and b are global variables the function is passed. Here the values returned by the function are assigned to the variable c

```
    return 0;
```

```
}
```

Function Definition

- contains the function statement return(x * y); the function returns x times y to the main function where it was called. Curly braces { } mark the start and end of the function

```
int product(int x, int y)
```

```
{  
    return (x * y);
```

```
}
```

1 Type of user defined function

1.3 Type of user defined function

➤ Type of user defined function :

1. No arguments passed and no return Value.
2. No arguments passed but a return value.
3. Argument passed but no return value
4. Argument passed and a return value .

I.No arguments passed and no return Value:

- The fun() function contain variable x and displays it on the screen.
- The empty parentheses in fun(); statement inside the main() function indicates that no argument is passed to the function.
- The return type of the function is void. Hence, no value is returned from the function .

1

How user defined function

1.2 How user defined function

CODE

```
void fun( );//prototype function or declaration function
// this function return void and take input integer

int main(void)
{
    int x = 20;
    fun();//call function by value

    printf("x = %d", x);
    return 0;
}

void fun();//definition function
{
    int x = 30;
    printf("%d\n", x);
    return;//this function return nothing
}
```

OUTPUT

```
30
x = 20
Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

DESCRIPTION

- In this example the function return void and print on screen variable x

1

Type of user defined function

1.3 Type of user defined function

➤ The return statement :

- terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

2.No arguments passed and return Value:

- The fun() function contain variable x and return variable x
- The empty parentheses in fun(); statement inside the main() function indicates that no argument is passed to the function.
- The return type of the function is int . Hence, value is returned from the function .

1

How user defined function

1.2 How user defined function

CODE

```
#include<stdio.h>
int fun( );//prototype function or declaration function
// this function return void and take input integer


int main(void)
{
    int x = 20;
    int y=fun();//call function by value

    printf("y= %d", y);
    return 0;
}

int fun()//definition function
{
    int x = 30;

    return x ;//this function return x
}
```

OUTPUT

y= 30

DESCRIPTION

- In this example the function return int and print returning on screen variable x

1

Type of user defined function

1.3 Type of user defined function

3. arguments passed but no return Value:

- The fun(int x) function contain variable x and return void
- The parentheses include integer x in fun(); statement inside the main() function indicates that argument is passed to the function.
- The return type of the function is void . Hence, no value is returned from the function .

1

How user defined function

1.2 How user defined function

CODE

```
#include<stdio.h>
void fun(int );//prototype function or declaration function
// this function return void and take input integer

int main(void)
{
    int x = 20;
    fun(x);//call function by value

    printf("x= %d", x);
    return 0;
}

void fun(int x)//definition function
{
    printf("x:%d\n",x);

    return ;//this function return nothing
}
```

OUTPUT

```
x:20
x= 20
```

DESCRIPTION

- In this example the function return void and print variable x on screen variable.

1

Type of user defined function

1.3 Type of user defined function

4. arguments passed and return Value:

- The fun(int x) function contain variable x and return integer.
- int fun(int x); statement inside the main() function indicates that argument is passed to the function and return int
- The return type of the function is int . Hence, value is returned from the function .

1

How user defined function

1.2 How user defined function

CODE

```
#include<stdio.h>
int fun(int );//prototype function or declaration function
// this function return void and take input integer

int main(void)
{
    int x = 20;
    int y=fun(x);//call function by value

    printf("y= %d", y);
    return 0;
}
int fun(int x)//definition function
{

    return x ;//this function return x
}
```

OUTPUT

```
y= 20
Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
```

DESCRIPTION

- In this example the function return int and take arguments integer.

1 Advantages of user defined function

1.4 Advantages of user defined function

➤ User defined functions in C programming has following advantages:

- 1. Reduction in Program Size:** Since any sequence of statements which are repeatedly used in a program can be combined together to form a user defined functions. And this functions can be called as many times as required. This avoids writing of same code again and again reducing program size.
- 2. Reducing Complexity of Program:** Complex program can be decomposed into small sub-programs or user defined functions.
- 3. Easy to Debug and Maintain :** During debugging it is very easy to locate and isolate faulty functions. It is also easy to maintain program that uses user defined functions.
- 4. Readability of Program:** Since while using user defined function, a complex problem is divided in to different sub-programs with clear objective and interface which makes easy to understand the logic behind the program.
- 5. Code Reusability:** Once user defined function is implemented it can be called or used as many times as required which reduces code repeatability and increases code reusability.

➤ What is Recursion?

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function, But while using recursion, programmers need to be careful to define an exit condition(Base condition) from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.
- Directly Call

```
// An example of direct recursion
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}
```

indirectly Call

```
// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
}
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

1 Recursive Function

1.5 Recursive Function

➤ **What is base condition in recursion?**

- In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.
- If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

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

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

1

Recursive Function

1.5 Recursive Function

CODE

```
// C code to implement factorial
#include <stdio.h>

// Factorial function
int f(int n)
{
    // Stop condition
    if (n == 0 || n == 1)
        return 1;

    // Recursive condition
    else
        return n * f(n - 1);
}

// Driver code
int main()
{
    int n = 5;
    printf("factorial of %d is: %d", n, f(n));
    return 0;
}
```

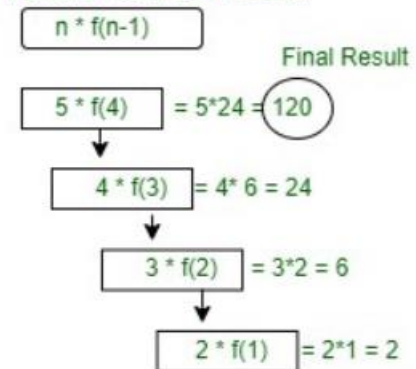
OUTPUT

factorial of 5 is: 120

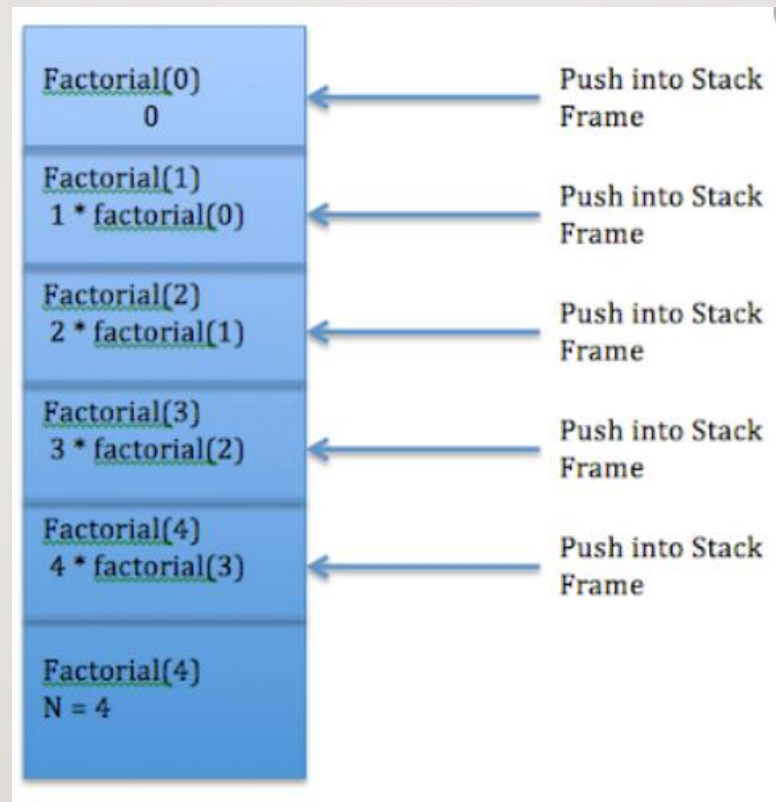
DESCRIPTION

For user input : 5

Factorial Recursion Function

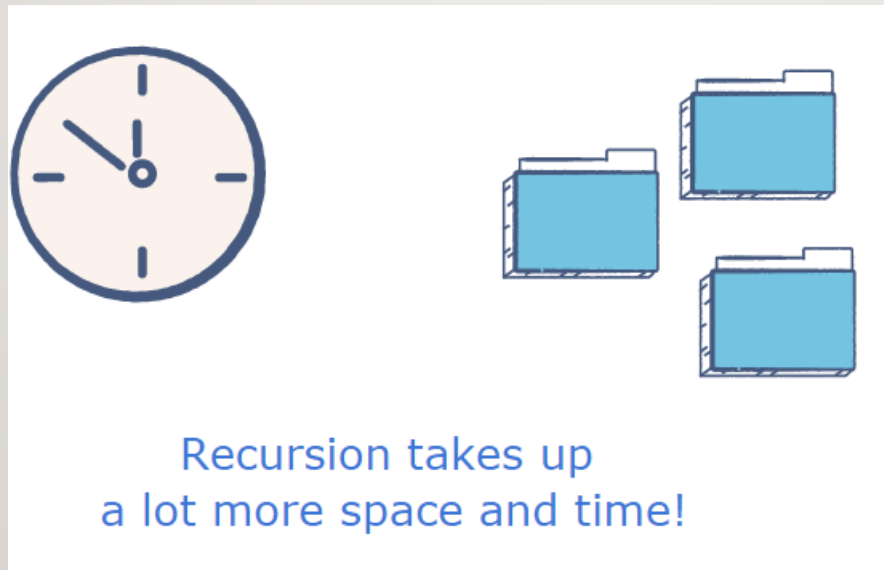


- How memory is allocated to different function calls in recursion?



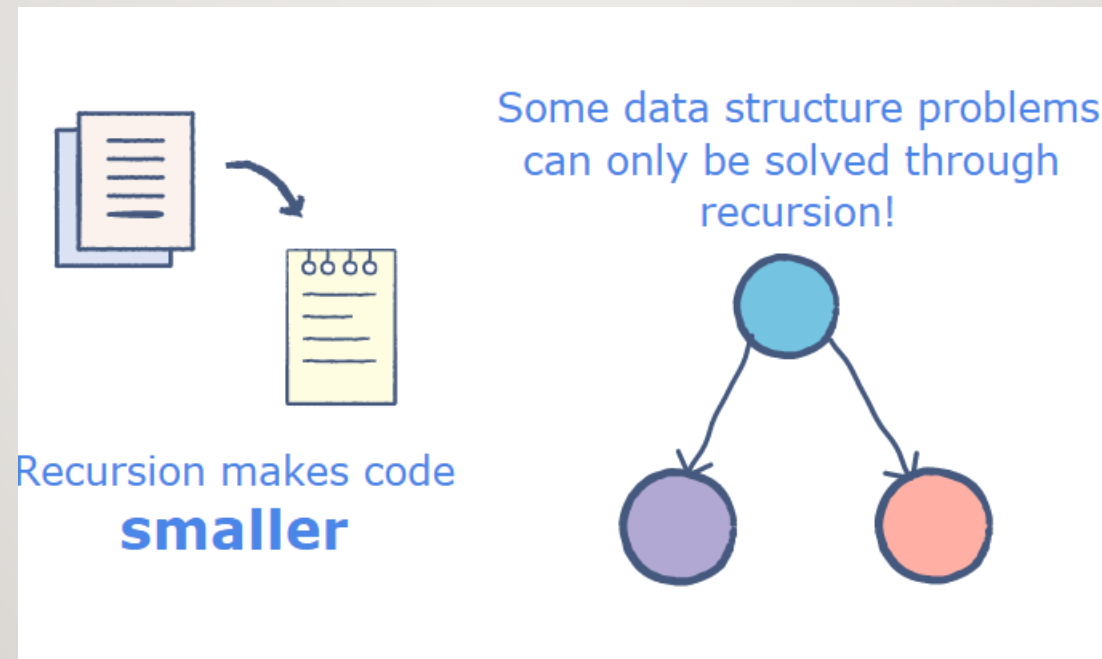
➤ Disadvantages of Recursion :

- A recursive program has greater space requirements than an iterative program as each function call will remain in the stack until the base case is reached.
- It also has greater time requirements because each time the function is called, the stack grows and the final answer is returned when the stack is popped completely.



➤ Advantages of Recursion :

- For a recursive function, you only need to define the base case and recursive case, so the code is simpler and shorter than an iterative code. And any problem solved in recursion can be solved by iterative.



2 Scope and life time of variable

2.1 Scope Rules in c

Scope	Meaning
File Scope	Scope of a Identifier starts at the beginning of the file and ends at the end of the file. It refers to only those Identifiers that are declared outside of all functions. The Identifiers of File scope are visible all over the file Identifiers having file scope are global
Block Scope	Scope of a Identifier begins at opening of the block / '{' and ends at the end of the block / '}'. Identifiers with block scope are local to their block
Function Prototype Scope	Identifiers declared in function prototype are visible within the prototype
Function scope	Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to goto statement and both goto and label statement must be in same function

2 Scope and life time of variable

2.1 Scope Rules in c

➤ Example

File scope

```
// C program to illustrate the global scope

#include <stdio.h>

// Global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    printf("%d\n", global);
}

// main function
int main()
{
    printf("Before change within main: ");
    display();

    // changing value of global
    // variable from main function
    printf("After change within main: ");
    global = 10;
    display();
}
```

```
// filename: file1.c
```

```
int a;

int main(void)
{
    a = 2;
}
```

```
// filename: file2.c
// When this file is linked with file1.c, functions
// of this file can access a
```

```
extern int a;
```

```
int myfun()
{
    a = 2;
}
```

Block scope

```
int main()
{
    {
        int x = 10, y = 20;
        {
            // The outer block contains
            // declaration of x and
            // y, so following statement
            // is valid and prints
            // 10 and 20
            printf("x = %d, y = %d\n", x, y);
            {
                // y is declared again,
                // so outer block y is
                // not accessible in this block
                int y = 40;

                // Changes the outer block
                // variable x to 11
                x++;

                // Changes this block's
                // variable y to 41
                y++;

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

            // This statement accesses
            // only outer block's
            // variables
            printf("x = %d, y = %d\n", x, y);
        }
    }
    return 0;
}
```

x = 10, y = 20

x = 11, y = 41

x = 11, y = 20

2 Scope and life time of variable

2.1 Scope Rules in c

Function Prototype Scope

```
// C program to illustrate
// function prototype scope

#include <stdio.h>

// function prototype scope
//(not part of a function definition)
int Sub(int num1, int num2);

// file scope
int num1;

// Function to subtract
int Sub(int num1, int num2)
{
    return (num1-num2);
}

// Driver method
int main(void)
{
    printf("%d\n", Sub(10,5));
    return 0;
}
```

Function Scope

```
void func1()
{
    {
        // label in scope even
        // though declared later
        goto label_exec;

        label_exec;;
    }

    // label ignores block scope
    goto label_exec;
}

void funct2()
{
    // throwerror:
    // as label is in func1() not funct2()
    goto label_exec;
}
```


2 Scope and life time of variable

2.1 Scope Rules in c

- What if the inner block itself has one variable with the same name?
 - If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the point of the declaration by inner block.
- What about functions and parameters passed to functions?
 - A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.
- Can variables of the block be accessed in another subsequent block?
 - No, a variable declared in a block can only be accessed inside the block and all inner blocks of this block.
- For example, the following program produces a compiler error.

```
int main()
{
    {
        int x = 10;
    }
    {
        // Error: x is not accessible here
        printf("%d", x);
    }
    return 0;
}
```

2 Scope and life time of variable

2.1 Scope Rules in c

- What if the inner block itself has one variable with the same name?
 - If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the point of the declaration by inner block.
- What about functions and parameters passed to functions?
 - A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.
- Can variables of the block be accessed in another subsequent block?
 - No, a variable declared in a block can only be accessed inside the block and all inner blocks of this block.
- For example, the following program produces a compiler error.

```
int main()
{
    {
        int x = 10;
    }
    {
        // Error: x is not accessible here
        printf("%d", x);
    }
    return 0;
}
```

2 Scope and life time of variable

2.2 Automatic , External, Static , Register

Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

2 Scope and life time of variable

2.2 Automatic , External, Static , Register

Automatic

```
// declaring the variable which is to be made extern
// an initial value can also be initialized to x
int x;

void autoStorageClass()
{
    printf("\nDemonstrating auto class\n\n");

    // declaring an auto variable (simply
    // writing "int a=32;" works as well)
    auto int a = 32;

    // printing the auto variable 'a'
    printf("Value of the variable 'a'"
           " declared as auto: %d\n",
           a);

    printf("-----");
}
```

External

```
void externStorageClass()
{
    printf("\nDemonstrating extern class\n\n");

    // telling the compiler that the variable
    // z is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;

    // printing the extern variables 'x'
    printf("Value of the variable 'x'"
           " declared as extern: %d\n",
           x);

    // value of extern variable x modified
    x = 2;

    // printing the modified values of
    // extern variables 'x'
    printf("Modified value of the variable 'x'"
           " declared as extern: %d\n",
           x);

    printf("-----");
}
```

2 Scope and life time of variable

2.2 Automatic , External, Static , Register

Static

```
void staticStorageClass()
{
    int i = 0;

    for (i = 1; i < 5; i++) {

        // Declaring the static variable 'y'
        static int y = 5;

        // Declare a non-static variable 'p'
        int p = 10;

        // Incrementing the value of y and p by 1
        y++;
        p++;

        // printing value of y at each iteration
        printf("\nThe value of 'y', "
               "declared as static, in %d "
               "iteration is %d\n",
               i, y);

        // printing value of p at each iteration
        printf("The value of non-static variable 'p', "
               "in %d iteration is %d\n",
               i, p);
    }
}
```

Register

```
void registerStorageClass()
{

    printf("\nDemonstrating register class\n\n");

    // declaring a register variable
    register char b = 'G';

    // printing the register variable 'b'
    printf("Value of the variable 'b'"
           " declared as register: %d\n",
           b);

    printf("-----");
}
```

2 Scope and life time of variable

2.2 Lab

➤ What will be the output of the program?

```
#include<stdio.h>
int i;
int fun();

int main()
{
    while(i)
    {
        fun();
        main();
    }
    printf("Hello\n");
    return 0;
}
int fun()
{
    printf("Hi");
}
```


3 Modular programming

3 Modular programming

- What is modular programming.
- What is Header Files.
- Large Scale projects.
- Data Sharing – extern keyword.
- Data Hiding – static keyword.

- What is modular programming:
 - Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
 - Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components.
 - Modules are typically incorporated into the program through interfaces. A module interface expresses the elements that are provided and required by the module.
 - The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface.

3 Modular programming

3 Modular programming

➤ What is modular programming:

- Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components.
- Modules are typically incorporated into the program through interfaces. A module interface expresses the elements that are provided and required by the module.
- The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface.

➤ What is Header files:

- Header files are files that are included in other files prior to compilation by the C preprocessor.
- Some, such as `stdio.h`, are defined at the system level and must be included by any program using the standard I/O library.
- Header files are also used to contain data declarations and defines that are needed by more than one program.
- Header files should be functionally organized, i.e., declarations for separate subsystems or functions should be in separate header files.
- Also, if a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

file1.h

```
int add(int,int);
```

file2.h

```
#include "file1.h"
```

```
int sub(int,int);
```

main.c

```
#include "file1.h"  
#include "file2.h"
```

```
int main() {  
    /* code */  
    return 0;  
}
```

file1.h included twice!!

3 Modular programming

3 Header files

➤ Header files should not be nested so must be **Header guards**

- It is common to put the following into each .h file to prevent accidental double-inclusion.

```
#ifndef EXAMPLE_H  
#define EXAMPLE_H  
/* body of example.h file */  
#endif /* EXAMPLE_H */
```

- This double-inclusion mechanism should not be relied upon, particularly to perform nested includes.

➤ Content header file:

- Function and type declarations, global variables, structure declarations and in some cases, inline functions,extern.
- definitions which need to be centralized in one file.
- In a header file, do not use redundant or other header files; only minimal set of statements.
- Don't put function definitions in a header. Put these things in a separate .c file.
- Include Declarations for functions and variables whose definitions will be visible to the linker.Also, definitions of data structures and enumerations that are shared among multiple source file

3 Modular programming

3 Large scale project

➤ Large Scale Project:

- Large projects may potentially involve many hundreds of source files (modules).
- Global variables and functions in one module may be accessed in other modules.
- Global variables and functions may be specifically hidden inside a module.
- Maintaining consistency between files can be a problem.

- Data Sharing

```
extern float step;

void print_table(double, float);

int main(void)
{
    step = 0.15F;

    print_table(0.0, 5.5F);

    return 0;
}
```

```
#include <stdio.h>

float step;

void print_table(double start, float stop)
{
    printf("Celsius\tFahrenheit\n");
    for(; start < stop; start += step)
        printf("%.11f\t%.11f\n", start,
                start * 1.8 + 32);
}
```

- Data Hiding: When static is placed before a global variable, or function, the item is locked into the module.

```
static int entries[S_SIZE];
static int current;

void push(int value)
{
    entries[current++] = value;
}

int pop(void)
{
    return entries[--current];
}

static void print(void)
{
}
```

```
void push(int value);
int pop(void);
void print(void);
extern int entries[];

int main(void)
{
    push(10); push(15);
    printf("%i\n", pop());

    entries[3] = 77;
    print();
    return 0;
}
```

3

Modular programming

3 Large scale project

➤ Recommendations to follow in large scale project:

- ✓ Use Header Files
- ✓ Maintain consistency between modules by using header files.
- ✓ Place Module definitions and configurations.
- ✓ Place an extern declaration of a variable in Module.c in a Module.h file.
- ✓ Place a prototype of a non static (i.e. Sharable) function in a Module.h file.
- ✓ Place the user defined data type declaration like structures, unions and enums declaration in a Module.h

Lab2

➤ Design a program composed of the following:

- Math Library:
MathLib.c and MathLib.h
`unsigned char Math_Add(unsigned char param_1, unsigned char param_2);`
- Error Log:
ErrorLog.c and ErrorLog.h
`void ErrLog_Log(void);`
- Main:
Main.c
Takes 2 8-bit variables from the user
Call `Math_Add()`
Detect if the value returned is a valid addition value or not
If the value is invalid, then report the error through calling `ErrLog_Log()`

Hint: Error can be caused due to addition overflow