CE143: COMPUTER CONCEPTS & PROGRAMMING

UNIT-9
User-Defined Function in 'C'

N. A. Shaikh
nishatshaikh.it@charusat.ac.in

Topics to be covered

- **Need of modularization & Advantages**
- **Introduction to user-defined function**
- **Function Prototype, Function Call, Function Body**
- **Call by value, Actual &Formal Arguments, return value**
- **Categories of functions**
- **Nesting of Functions**
- **Recursion**
- **Array as Function arguments**
- **Storage Classes: Scope, Life of a variable in ‘C’.**

Need of Modularization in C

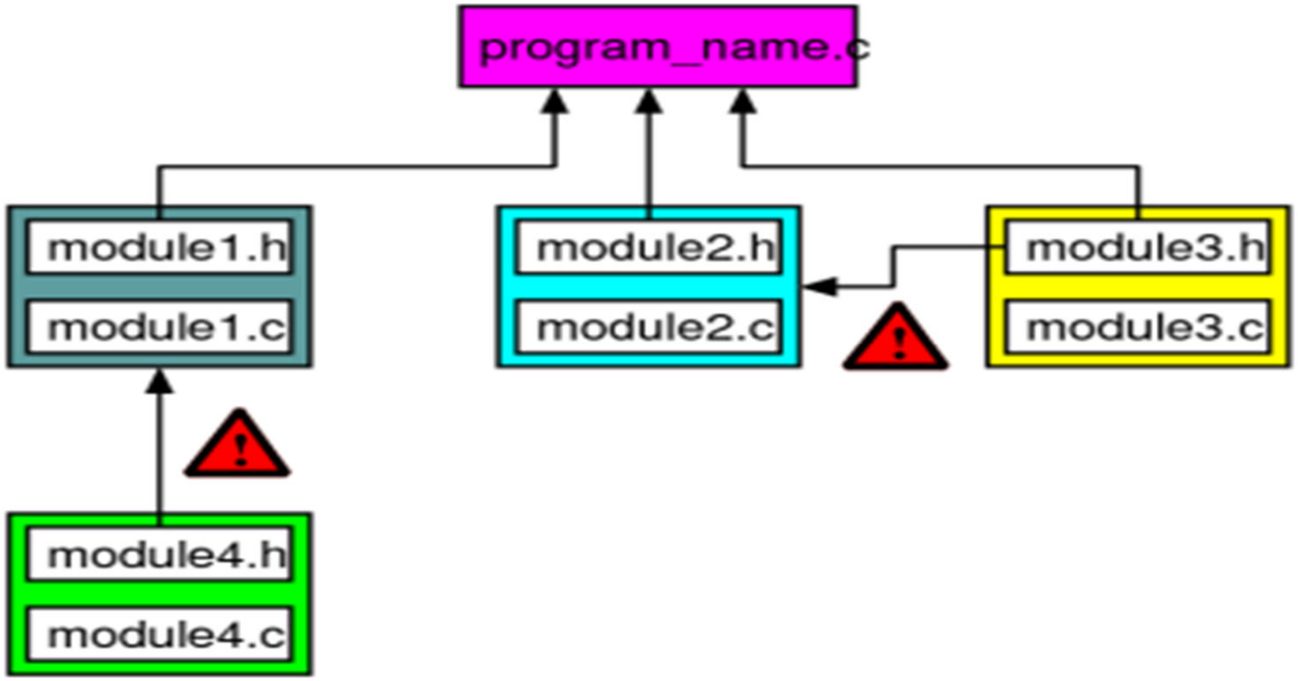
Modularization is a method to organize large programs in smaller parts, i.e. the modules.

Modular programming is the process of subdividing a computer program into separate sub-programs.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program.
- So, to manage such type of programs, **concept of modular programming approached.**

Need of Modularization in C

- Modular programming emphasis on **breaking of large programs into small problems** to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.



Advantages of Modularization in C

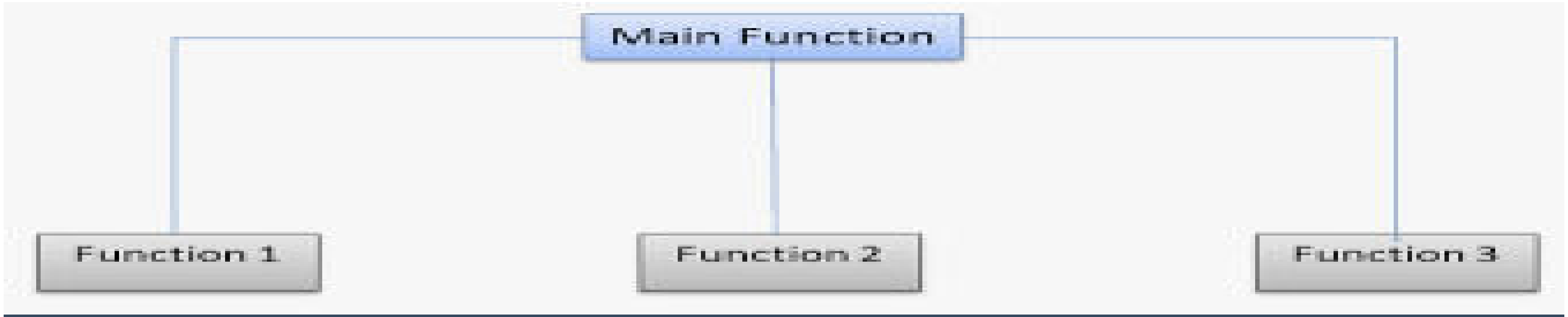
Ease of Use : This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.

Reusability : It allows the user to reuse the functionality with a different interface without typing the whole program again.

Ease of Maintenance : It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

Introduction

- As we discussed , large and complex programs will make task of debugging, testing and maintaining difficult.
- If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit.
- These independently coded programs are called subprograms that are much easier to understand , debug and test.
- In C, such subprograms are referred to as **‘FUNCTIONS’**
- This saves both time and space



Function

- **A function is a set of statements that take inputs, do some specific computation and produces output.**
- **Also known as procedure, subroutine or subprograms**
- Function can be treated as a **'black box'** that take some data from the main program and returns a value.
- The inner details of operation are invisible to the rest of the program. All that program knows about a function is: **What goes in and what comes out.**
- Every C program can be designed using a collection of these black boxes known as functions.
- 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.

Function

- Functions are classified as one of the **derived data types** in C
- Every C program has at least one function, which is **main()**.

Similarities between functions and variables

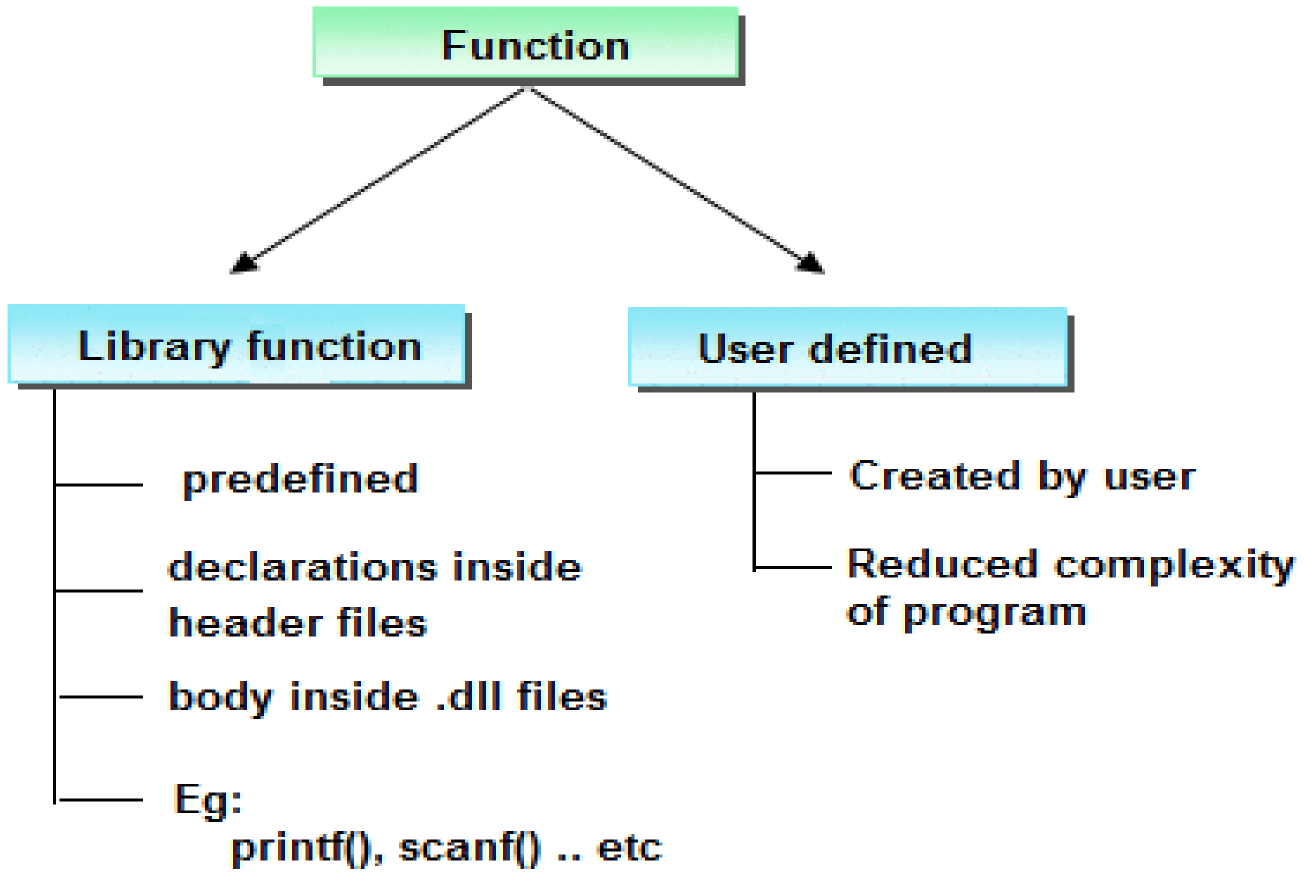
- Both function names and variable names are considered identifiers and therefore, they must adhere to the **rules for identifiers.**
- Like variables, functions have **data types** associated with them
- Like variables, **function names and their types must be declared** and defined before they are used in program.

Need of Function

- 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.

Types of Functions

C functions can be classified into two categories:



Function
Library function
User defined
predefined
declarations inside header files
body inside .dll files
Eg: printf(), scanf() .. etc
Created by user
Reduced complexity of program

Types of Functions

Library (Built-in) Function	User Defined Function
Functions which are already defined, compiled and stored in different header file of C Library are known as **Library Functions**.	Those functions which are defined by programmers according to their need are known as **User Defined Functions**.
These functions cannot be modified by user.	These functions can be modified by user.
Users do not know the internal working of these type of functions.	Users understand the internal working of these type of functions.

For Example: printf(), scanf(), getch(), clrscr(), pow() etc.	For Example:
	void message()
	{
	 printf("Welcome to C.");
	 }
	Here message() is user defined function.

Elements of User Defined Functions

1. Function definition/ Function implementation

2. Function call

3. Function Declaration/ Function Prototype

Elements of User Defined Functions

1. **Function definition/ Function implementation**

➢ The function definition is an independent program module that is specially written to implement the requirements of the function.

2. **Function call**

➢ In order to use this function we need to invoke it at a required place in the program. This is known as function call

3. **Function Declaration/ Function Prototype**

➢ The calling program should declare any function that is to be used later in the Program. This is known as Function Declaration or Function Prototype.

Function definition/ Function implementation

It includes the following elements:

1. Function return type
2. Function name
3. List of parameters (Formal parameters)
4. Local variable declarations
5. Function statements
6. A return statement

All the six elements are grouped into two parts, namely,

- Function header (First three elements)
- Function body (Second three elements)

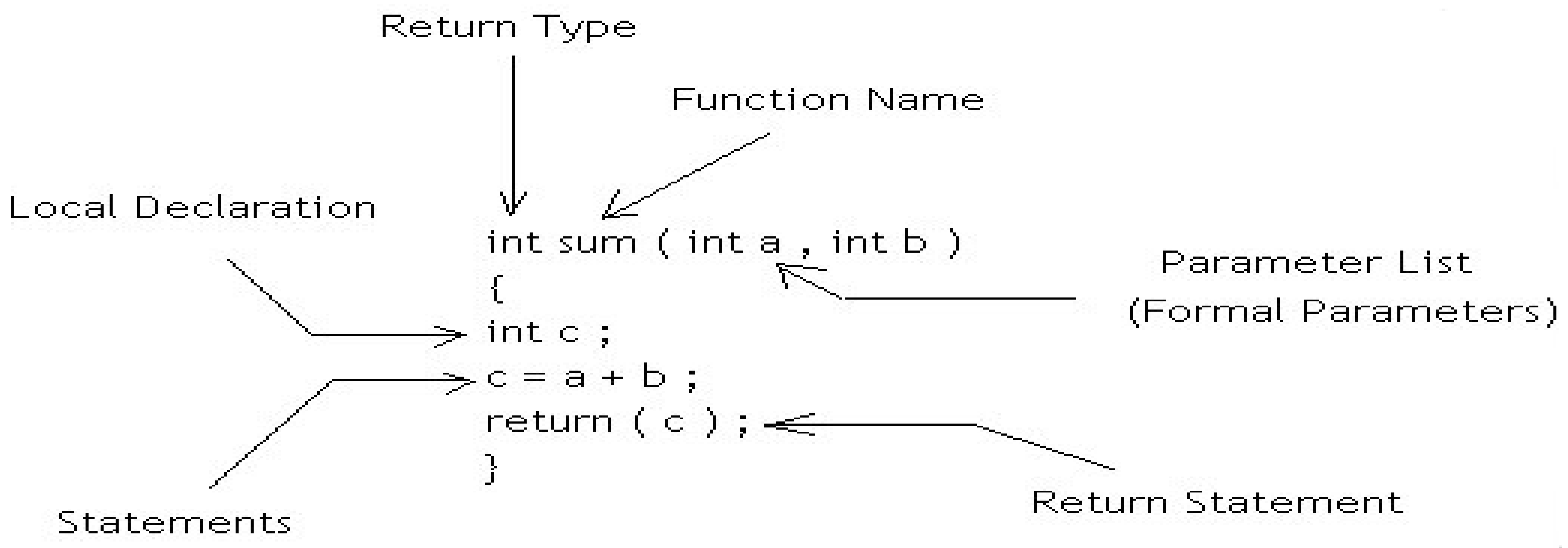
Function definition/ Function implementation

Syntax:

```
function_return_type function_name (parameter-list)
{
    local variable declaration;
    executable statement 1;
    executable statement 2;
    ........
    ........
    return statement;
}
```

Function definition/ Function implementation

Example:



```
int sum ( int a , int b )
{
int c ;
c = a + b ;
return ( c ) ;
}
```

Function definition/ Function implementation

Example:

```
float mul(float x,float y)
{
    float result;
    result=x*y;
    return (result);
}
```

Function Type:-

- Specifies the type of value that the function is expected to return to the program calling the function
- If not specified, C assumes it as **int** type
- If function is not returning anything, specify it as **void**

Function Name:-

- Any valid C identifier

Formal Parameter List:-

- Declares the variables that will receive the data sent by the calling program
- Separated by commas and surrounded by parenthesis

NOTE:

- No semicolon after the closing parenthesis
- float mul(float x,y) is **illegal**.

Local Declaration:-

- Specify the variables needed by the function

Function Statements:-

- Perform the task of function

Return statement:-

- Return the value evaluated by the function
- If a function does not return any value, we can omit return statement. **NOTE:** its return should be specified as void

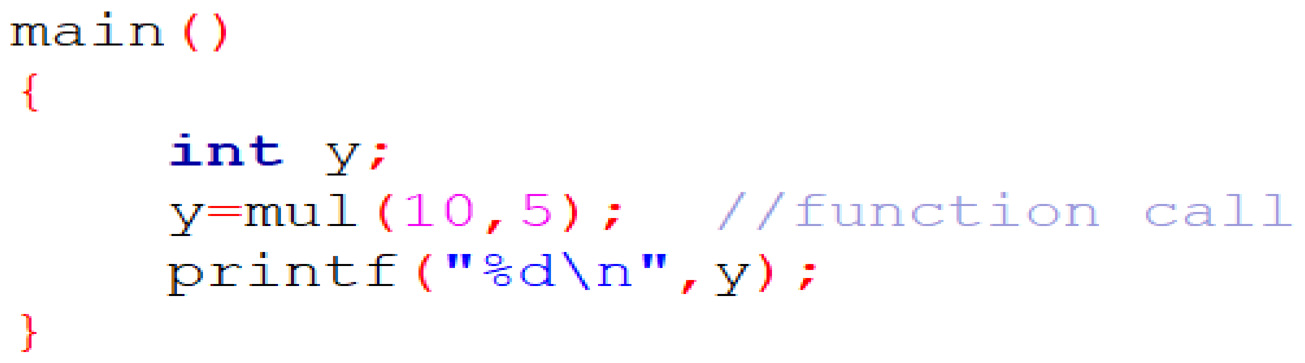
NOTE:

- When a function reaches its return statement, the control is transferred back to the calling program.
- In the absence of a return statement, the closing brace acts as a void return.

```
return; //does not return any value

OR

return(expression);
```

Function call

A function can be called by simply using the **function name** followed by a list of **actual parameters**



```
main()
{
        int y;
        y=mul(10,5);   //function call
        printf("%d\n",y);
}
```

- When compiler encounters a function call, the control is transferred to the function mul()
- mul() function is then executed line by line
- Value is retuned when a return statement is encountered
- This value is assigned to y.

Function call

Different ways to call a function:

```
mul(10,5);
mul(m,5);
mul(10,n);
mul(m,n);
mul(m+5,10);
mul(10,mul(m,n));
mul(exp1,exp2);
```

Function cannot be used on left side of assignment

```
mul(a,b)=15;
```

NOT VALID

Function Declaration/ Function Prototype

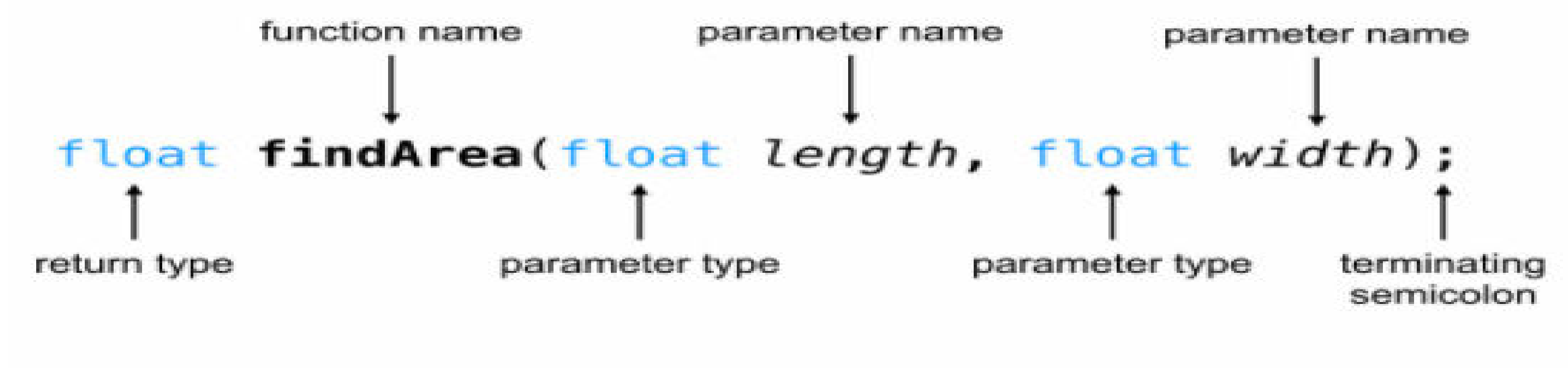
Like variables, all functions must be declared, before they are invoked.

Syntax:

```
Function-return-type function-name (parameter-list);
```

Example:

```
int mul (int m, int n);
```



Function Declaration/ Function Prototype

NOTE:

1. The parameter list must be separated by **commas**.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must **match the types of parameters** in the function definition, in number and order.
4. Use of parameter names in the declaration is **optional**.
5. If the function has no formal parameters, the list is written as (**void**)
6. The return type is optional, when the function returns **int** type data.
7. The retype must be **void** if no values is returned
8. When the declared types do not match with the types in the function definition, compiler will produce an error.

Function Declaration/ Function Prototype

Acceptable forms of declaration of mul function:

```
int mul (int, int);
    mul (int a, int b);
    mul (int, int);
```

When a function does not take any parameters and does not return any value,

```
void display (void);
```

Function Declaration VS Function Definition

Function Declaration	Function Definition
A prototype that specifies the function name, return types and parameters without the function body	Actual function that specifies the function name, return types and parameter with the function body
can be declared **any number of times**	can be defined only **once**
Memory will not be allocated during declaration	**Memory will be allocated**

int f(int);	int f(int a)
	{
	  return a;
	 }

The above is a function declaration. This declaration is just for informing the compiler that a function named f with return type and argument as int will be used in the function.	Helps to write what the function should perform. It is the actual implementation of the function

Function Parameters/ Function Arguments

Used in following three places:

1. In function declaration (prototypes)
2. In function call
3. In function definition

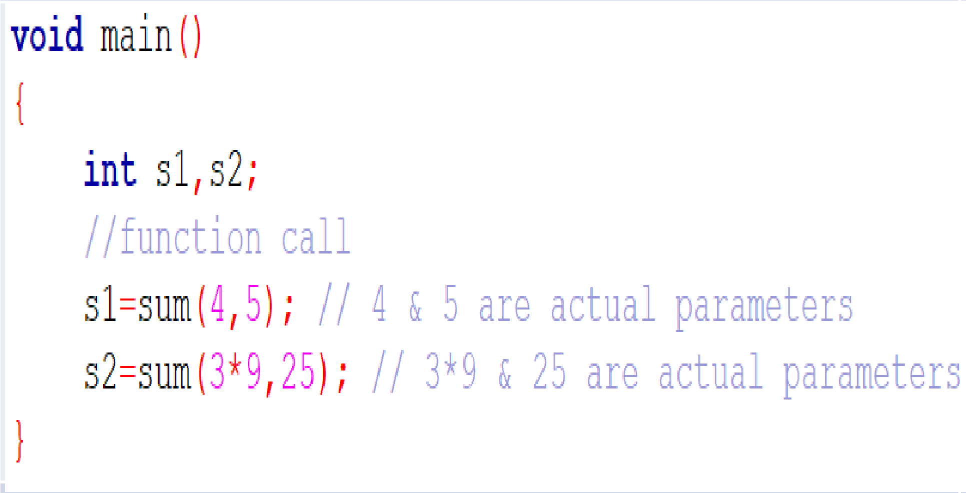
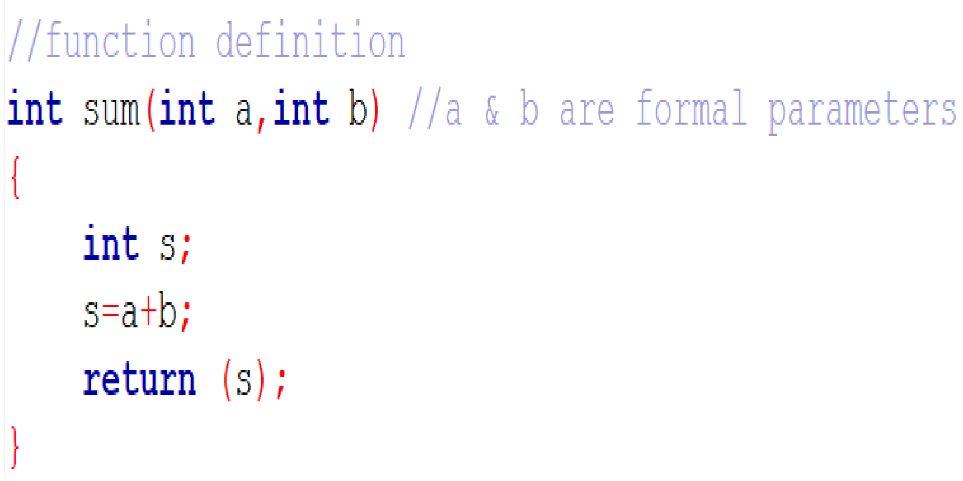
Actual Parameters: Parameters used in function call
Formal Parameters: Parameters used in function definition

NOTE:

- Actual parameters can be simple constants, variables or expressions
- The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

Actual Parameters VS Formal Parameters

Actual Parameters	Formal Parameters
These are ordered list of parameters which are included at the time of function call.	An ordered list of parameters which are included at the time of definition of function.
Data type not required. But data type should match with corresponding data type of formal parameters.	Data types needs to be mentioned.
These can be variables, expression and constant without data types.	These are variables with their data type.

 `void main()`
`{`
`    int s1,s2;`
`    //function call`
`    s1=sum(4,5); // 4 & 5 are actual parameters`
`    s2=sum(3*9,25); // 3*9 & 25 are actual parameters`
`}`
	 `//function definition`
	`int sum(int a,int b) //a & b are formal parameters`
	`{`
	`    int s;`
	`    s=a+b;`
	`    return (s);`
	`}`

These are actual values which are given as input to the called function.	These are only definition of inputs data types. These are to shows which type of data should be given as input while calling the function.

Actual Parameters VS Formal Parameters

NOTE:

- If the actual parameters are more than the formal parameters

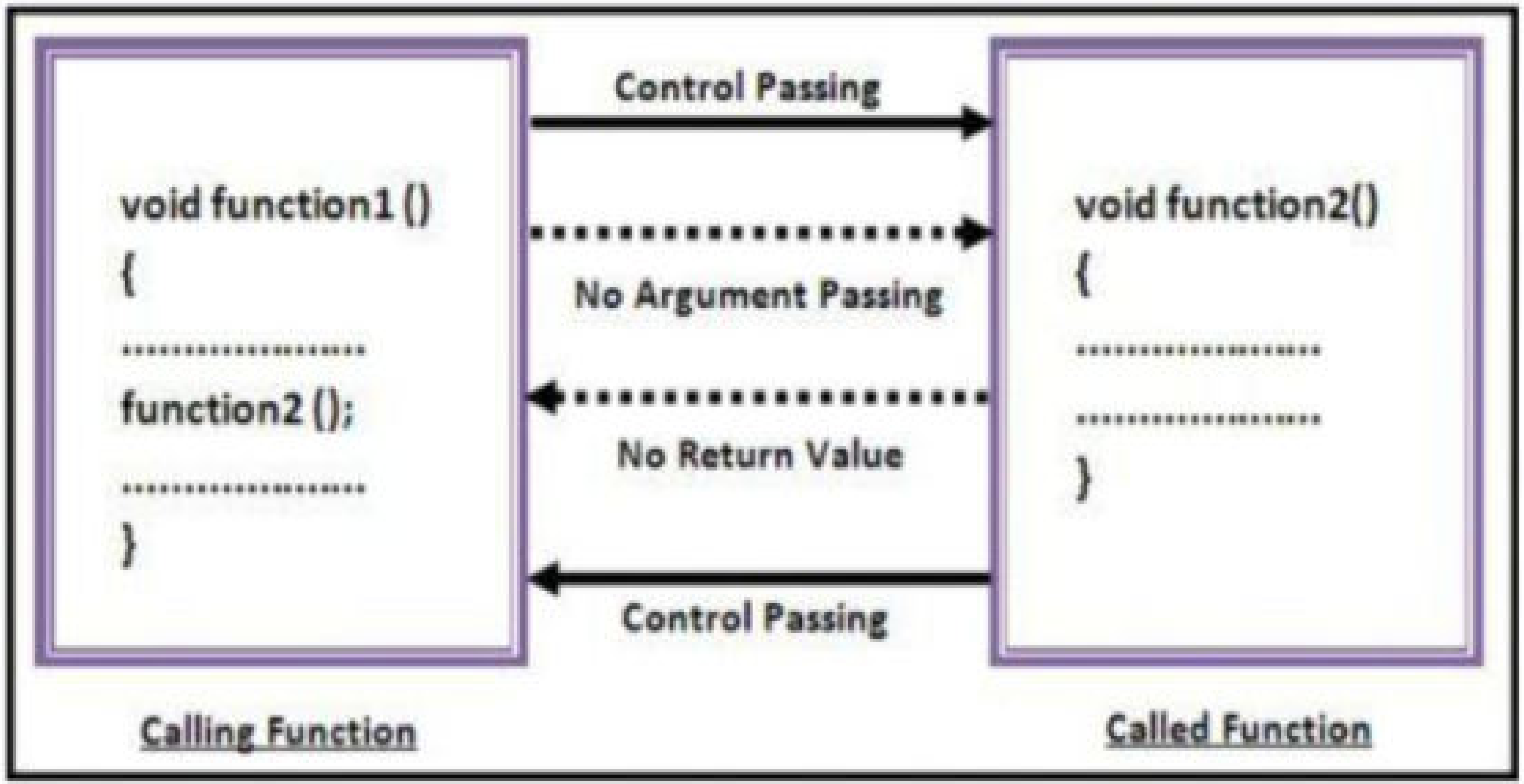
  error: too many arguments to function 'sum’
- On the other hand, if the actuals are less than the formals

  error: too few arguments to function 'sum’
- Any mismatch in data types may also result in some garbage values.

Category of Function

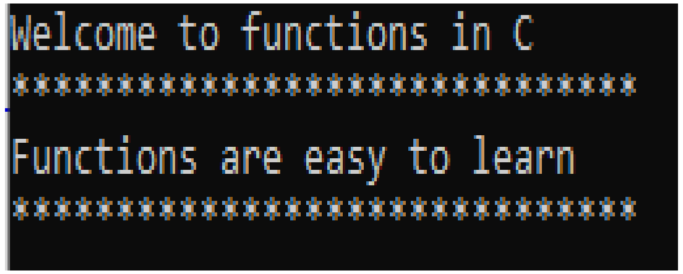
1. Functions with no arguments and no return values
2. Functions with arguments and no return value
3. Functions with arguments and one return value
4. Functions with no arguments but return a value
5. Functions that return multiple values

1) Functions with no arguments and no return values



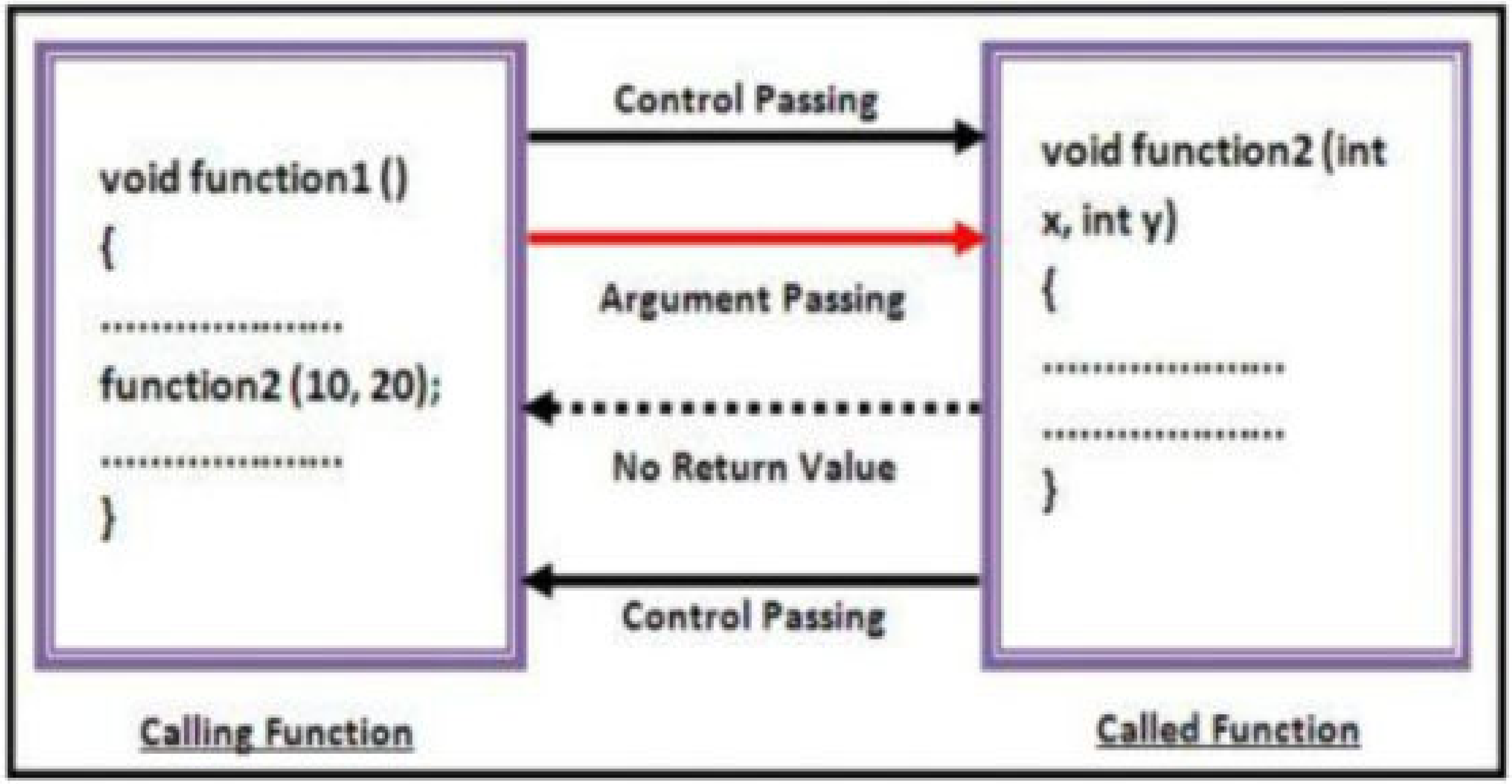
1) Functions with no arguments and no return values

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

void main()
{
     printf("Welcome to functions in C");
     println();
     printf("Functions are easy to learn");
     println();
}

void println()
{
     int i;
     printf("\n");
     for(i=0;i<30;i++)
          printf("*");
     printf("\n");
}
```



```
Welcome to functions in C
******************************

Functions are easy to learn
******************************
```

2) Functions with arguments and no return value

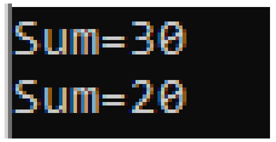


```
void function1 ()
{
....................
function2 (10, 20);
....................
}
```

```
void function2 (int
x, int y)
{
....................
....................
}
```

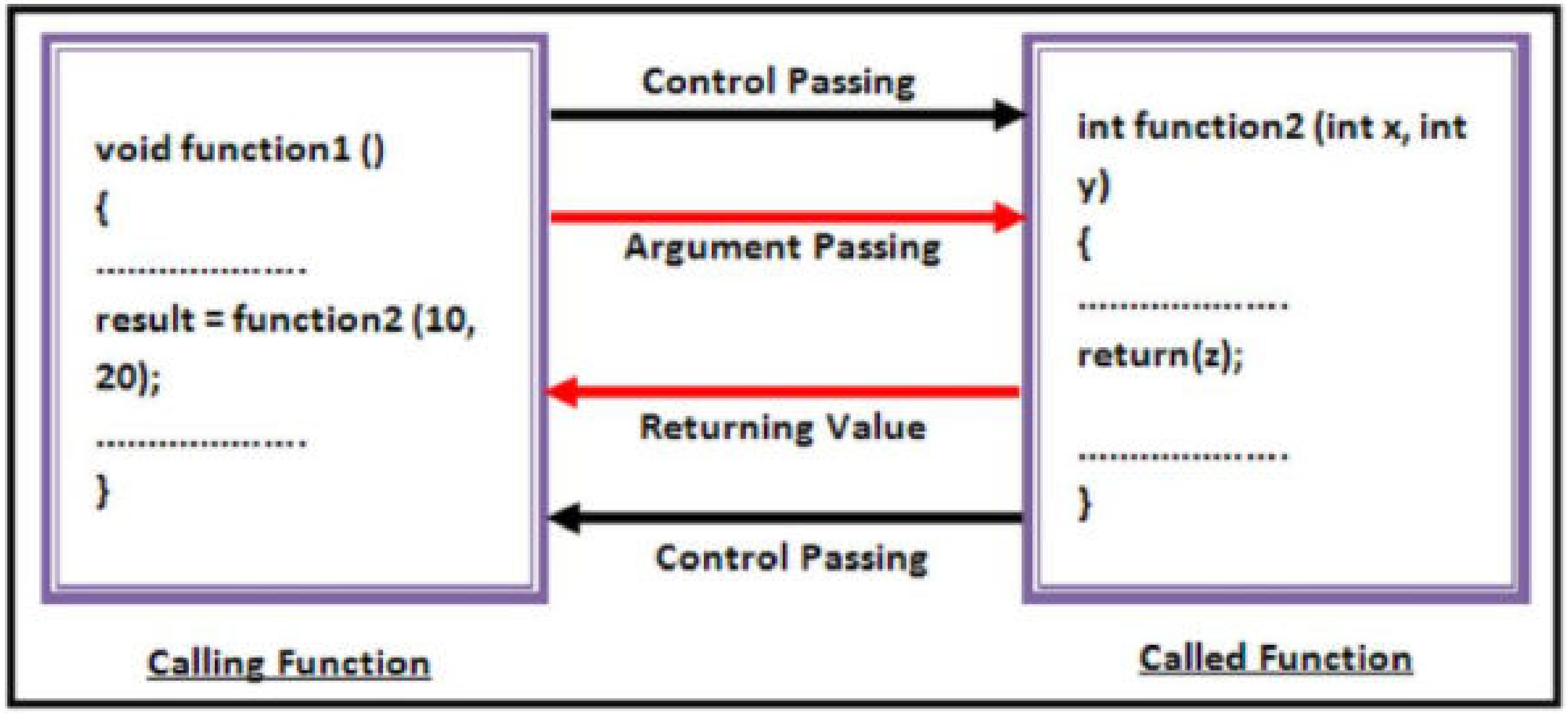
2) Functions with arguments and no return value

```
#include<stdio.h>
void add(int x,int y);

void main()
{
    int m=15,n=5;

    add(10,20); //actual parameters
    add(m,n);   //actual parameters
}

void add(int x,int y)  //formal parameters
{
    int result;
    result=x+y;
    printf("Sum=%d\n",result);
    return;  //Optional
}
```



```
Sum=30
Sum=20
```

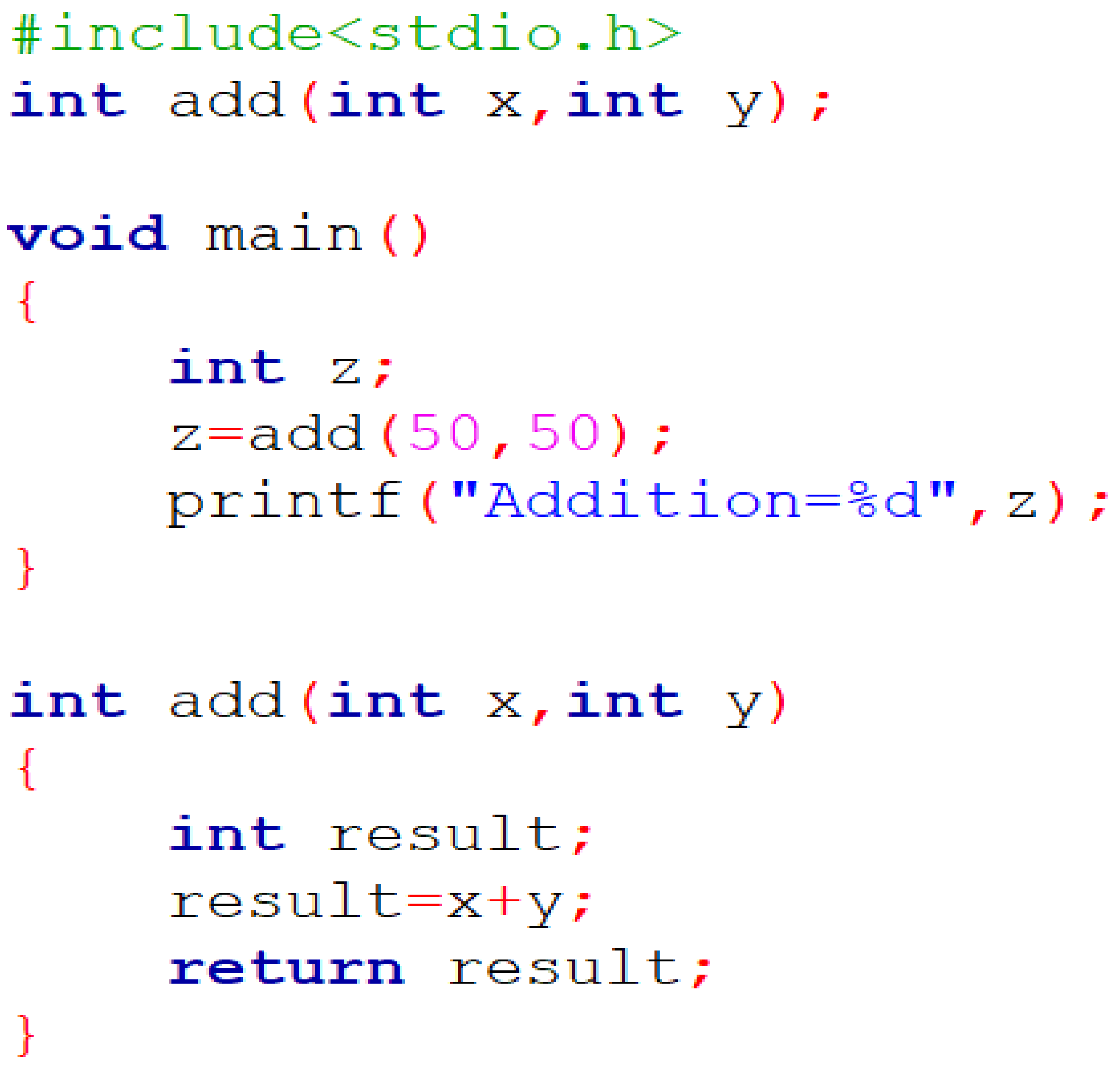
3) Functions with arguments and one return value



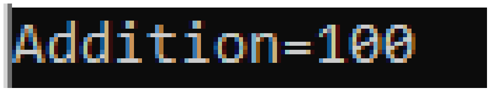
```
void function1 ()
{
......................
result = function2 (10,
20);
......................
}
```

```
int function2 (int x, int
y)
{
......................
return(z);

......................
}
```

3) Functions with arguments and one return value

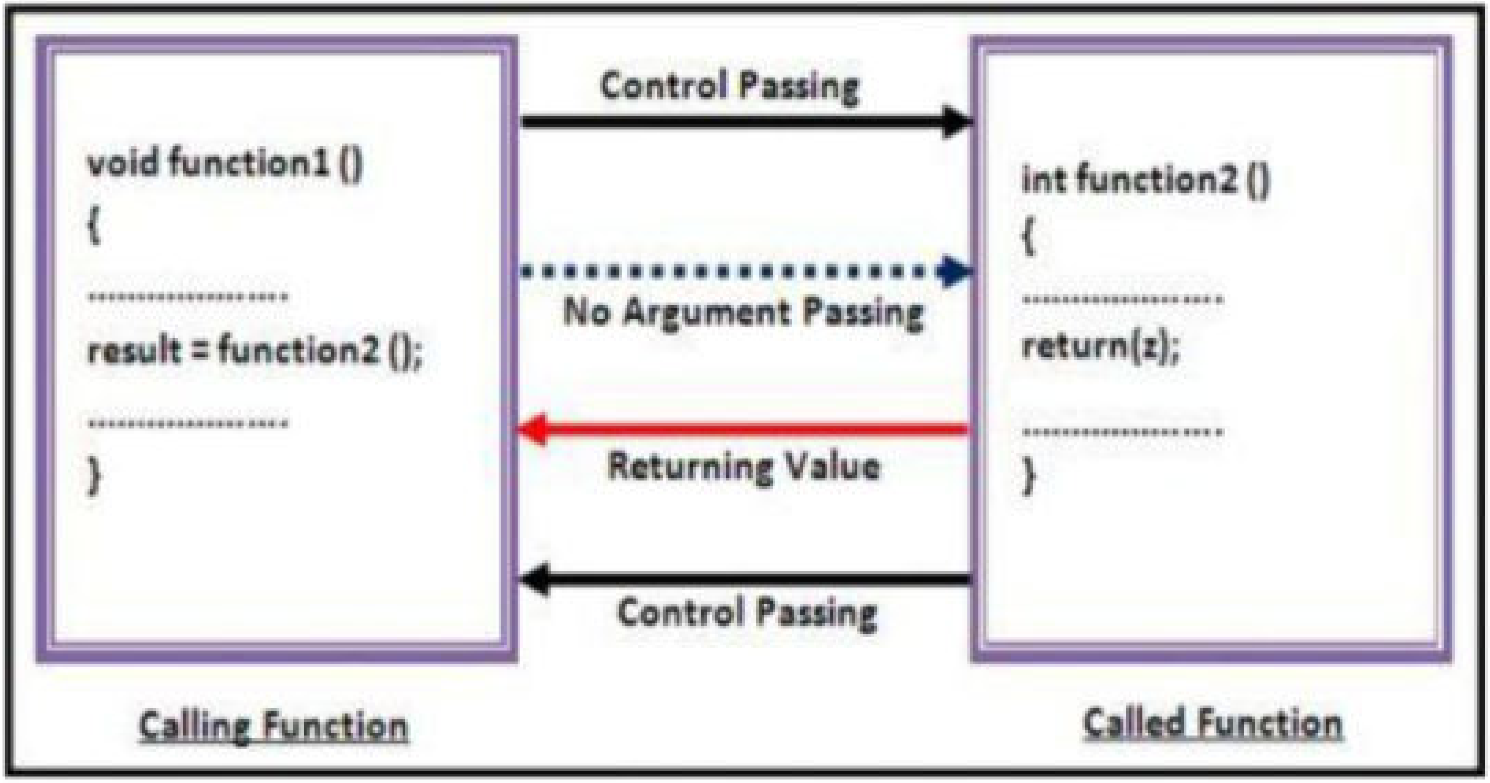


```
#include<stdio.h>
int add(int x,int y);

void main()
{
    int z;
    z=add(50,50);
    printf("Addition=%d",z);
}

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



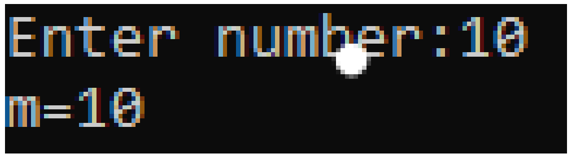
```
Addition=100
```

4) Functions with no arguments but return a value



4) Functions with no arguments but return a value

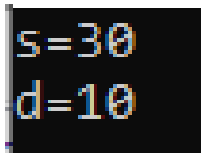
```c
#include<stdio.h>
int get_number(void);

void main()
{
      int m=get_number();
      printf("m=%d",m);
}

int get_number(void)
{
      int number;
      printf("Enter number:");
      scanf("%d",&number);
      return number;
}
```



```
Enter number:10
m=10
```

5) Functions that return multiple values

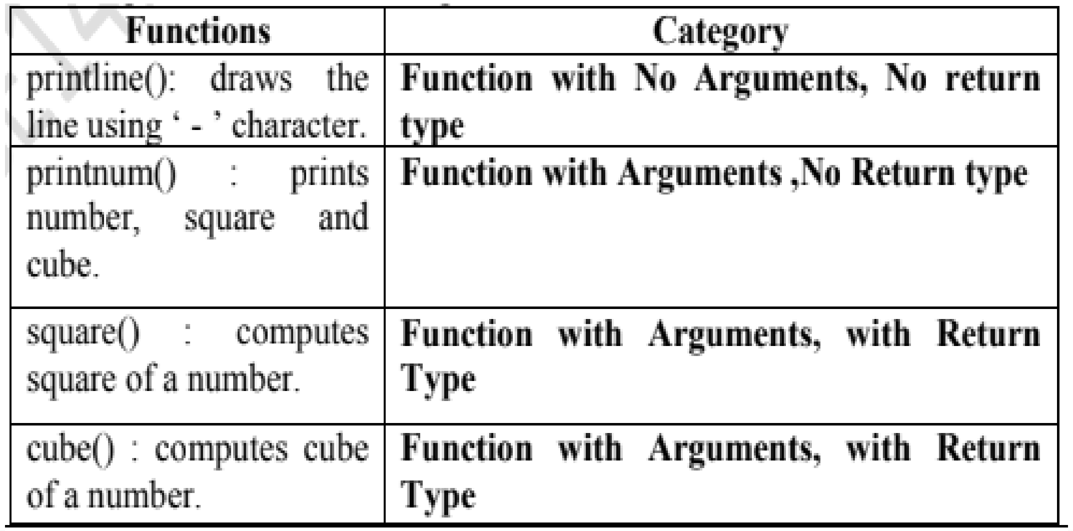
```c
#include<stdio.h>
int calc(int x,int y,int *s,int *d);

void main()
{
    int x=20,y=10,s,d;
    calc(x,y,&s,&d);
    printf("s=%d\n d=%d\n",s,d);
}

int calc(int a,int b,int *sum,int *diff)
{
    *sum=a+b;
    *diff=a-b;
}
```



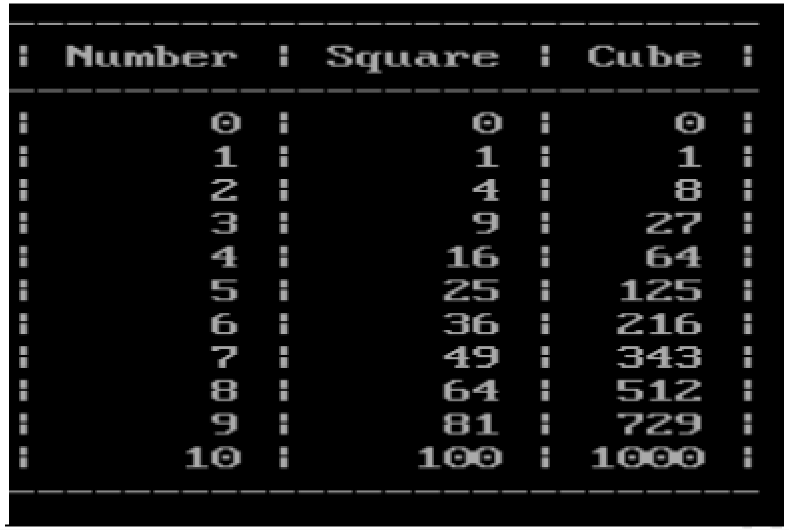
```
s=30
d=10
```

Category of Function

Write a Program to print the table of Squares and Cubes of 0 to 10.
The Program uses the following four Function:



Functions	Category
printline(): draws the line using ‘ - ’ character.	**Function with No Arguments, No return type**
printnum() : prints number, square and cube.	**Function with Arguments ,No Return type**
square() : computes square of a number.	**Function with Arguments, with Return Type**
cube() : computes cube of a number.	**Function with Arguments, with Return Type**



```
------------------------------------------
| Number  | Square  |  Cube  |
------------------------------------------
|       0 |       0 |      0 |
|       1 |       1 |      1 |
|       2 |       4 |      8 |
|       3 |       9 |     27 |
|       4 |      16 |     64 |
|       5 |      25 |    125 |
|       6 |      36 |    216 |
|       7 |      49 |    343 |
|       8 |      64 |    512 |
|       9 |      81 |    729 |
|      10 |     100 |   1000 |
------------------------------------------
```

What happens when a function is called before its declaration in C?

- In C, if a function is called before its declaration, the **compiler assumes return type of the function as int**.

```
#include <stdio.h>
void main()
{
    // Note that fun() is not declared
    printf("%d\n", fun());
}

char fun()
{
   return 'G';
}
```

Output : Program fails in compilation.

```
warning: implicit declaration of function 'fun'
error: conflicting types for 'fun'|
```

NOTE: If the function char fun() in above code is defined before main() then it will compile and run perfectly.

What happens when a function is called before its declaration in C?

```c
#include <stdio.h>
void main()
{
    // Note that fun() is not declared
    printf("%d\n", fun());
}

int fun()
{
   return 'D';
}
```

Output:

```
68
```

warning: implicit declaration of function 'fun'

What happens when a function is called before its declaration in C?

What about parameters?

- compiler assumes nothing about parameters. Therefore, the compiler will not be able to perform compile-time checking of argument types and arity when the function is applied to some arguments. This can cause problems. For example, the following program compiled fine in GCC and produced garbage value as output.

```
#include <stdio.h>

int main (void)
{
    printf("%d", sum(10, 5));
    return 0;
}
int sum (int b, int c, int a)
{
    return (a+b+c);
}
```

NOTE: There is this misconception that the compiler assumes input parameters also int. Had compiler assumed input parameters int, the above program would have failed in compilation.

Practical-9.1

Write a C program to check if the entered number is prime or not by using types of user defined functions

(i) No arguments passed and no return value
(ii) No arguments passed but a return value
(iii) Argument passed but no return value
(iv) Argument passed and a return value

Practical-9.2

If the length of the sides of a triangle are denoted by a, b and c, then find the area of triangle using function:

$$s = \frac{a+b+c}{2}$$

$$A = \sqrt{s(s-a)\times(s-b)\times(s-c)}$$

1. Define two user defined functions add() and mul() in a C Program which performs addition and multiplication of two numbers respectively. Write main() to call both the functions.
2. Write a program to find weather the entered number is odd or even using function (Use category with argument with return type)
3. Write a function program to sum of odd numbers between 1 to n, where n is entered through keyboard
4. Write a function power(X, n) that computes X^ n where X and n are integer values and return double type value.
5. Write a program to evaluate the series: 1! – 2! +3! – 4! ...± n!. Define a User-Defined function fact() that takes a number as an argument and returns its factorial.
6. Write a program to evaluate the series: $1 + \frac{2^2}{2!} + \frac{3^2}{3!} + \cdots + \frac{n^2}{n!}$. Define two user-defined functions fact ( ) to calculate factorial and square ( ) to calculate square of a number.
7. Write a program to check whether the entered number is Krishnamurti number or not. Use user-defined function factorial( ) to calculate factorial. (Krishnamurti number is a number which is equal to the sum of the factorials of all its digits.)

Parameter Passing to Function

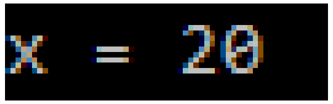
Two different ways of passing values to functions:

1. call by value (pass by value)
2. call by address (pass by address)

call by value (pass by value)

- **values of actual parameters are copied** to function's formal parameters
- parameters are stored in **different memory locations**
- So any **changes made inside functions are not reflected** in actual parameters of the caller.

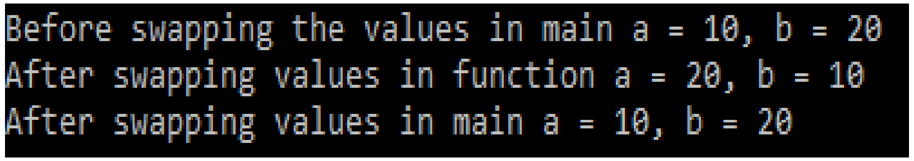
```c
#include <stdio.h>

void fun(int x)
{
     x = 30;
}

void main()
{
     int x = 20;
     fun(x);  //call by value
     printf("x = %d", x);
}
```



```
x = 20
```

call by value (pass by value)

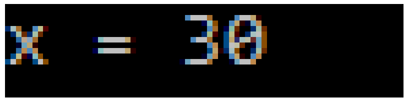
```c
#include <stdio.h>
void swap(int , int);

void main()
{
     int a = 10,b=20;
     printf("Before swapping the values in main a = %d, b = %d\n",a,b);
     swap(a,b);
     printf("After swapping values in main a = %d, b = %d\n",a,b);
}

void swap (int a, int b)
{
     int temp;
     temp = a;
     a=b;
     b=temp;
     printf("After swapping values in function a = %d, b = %d\n",a,b);
}
```



```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

call by address (pass by address)

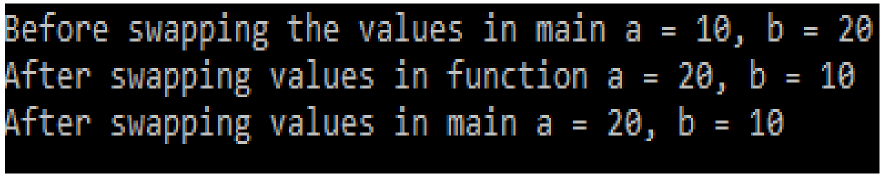
- **Address of actual parameters are passed** to function's formal parameters
- parameters are stored in **same memory locations**
- So any **changes made inside functions are reflected** in actual parameters of the caller.
-
```c
# include <stdio.h>

void fun(int *ptr)
{
     *ptr = 30;
}

void main()
{
     int x = 20;
     fun(&x);  //call by address
     printf("x = %d", x);
}
```



```
x = 30
```

call by address (pass by address)

```c
#include <stdio.h>
void swap(int *, int *);

void main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b);
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b);
}

void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a=*b;
    *b=temp;
    printf("After swapping values in function a = %d, b = %d\n",*a,*b);
}
```



```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

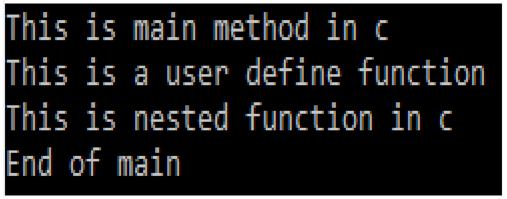
call by value VS call by address

Basis for Comparison	Call by Value	Call by address
Basic	A copy of actual parameters is passed into formal parameters.	address of actual parameters is passed into formal parameters.
Effect	Changes in formal parameters will not result in changes in actual parameters	Changes in formal parameters will result in changes in actual parameters.
Memory Allocation	Separate memory location is allocated for actual and formal parameters.	Same memory location is allocated for actual and formal parameters.
Calling Parameters	function_name(variable_name1, variable_name2, ...);	function_name(&variable_name1, &variable_name2, ...);
Receiving Parameters	type function_name(type variable_name1, type variable_name2,. . . .) { . . }	type function_name(type *variable_name1, type *variable_name2, ...) { . . }
Passing of variable	variables are passed using a straightforward method	pointers are required to store the address of variables.

Nesting of function

C permits nesting of functions freely.

```
#include <stdio.h>

void function_two()
{
    printf("This is nested function in c\n");
}
void function_one()
{
    printf("This is a user define function\n");
    function_two();
}

void main()
{
    printf("This is main method in c\n");
    function_one();
    printf("End of main\n");

}
```



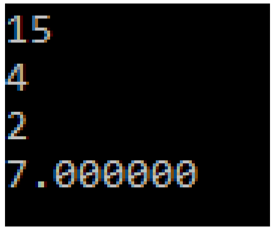
```
This is main method in c
This is a user define function
This is nested function in c
End of main
```

Nesting of function is also possible in following way(two sequential function calls)

```
p=mul(mul(5,2),6);
```

Nesting of function

```c
#include <stdio.h>
float ratio(int x,int y,int z);
int difference(int x,int y);

void main()
{
	int a,b,c;
	scanf("%d %d %d",&a,&b,&c);
	printf("%f\n",ratio(a,b,c));
}

float ratio(int x,int y,int z)
{
	if(difference(y,z))
		return (x/(y-z));
	else
		return (0.0);
}

int difference(int p,int q)
{
	if(p!=q)
		return 1;
	else
		return 0;
}
```



```
15
4
2
7.000000
```

Nesting of function

Write a program to pass a number entered through keyboard as an argument to user-defined functions and find the factors of a number and check whether the factors are prime or not using Nested Functions.

Recursion

- When a called function in turn calls another function a process of ‘**chaining**’ occurs.
- **Recursion** is a special case of this process, where a **function calls itself**.
- A function calling itself is called **recursive function** and process is known as **Recursion**

```
void main()
{
    printf("This is an example of recursion\n");
    main(); //recursive function
}
```

Execution is terminated abruptly or continue indefinitely

There must be a **terminating condition** to stop the recursion. This condition is known as the **base condition**.

Recursion

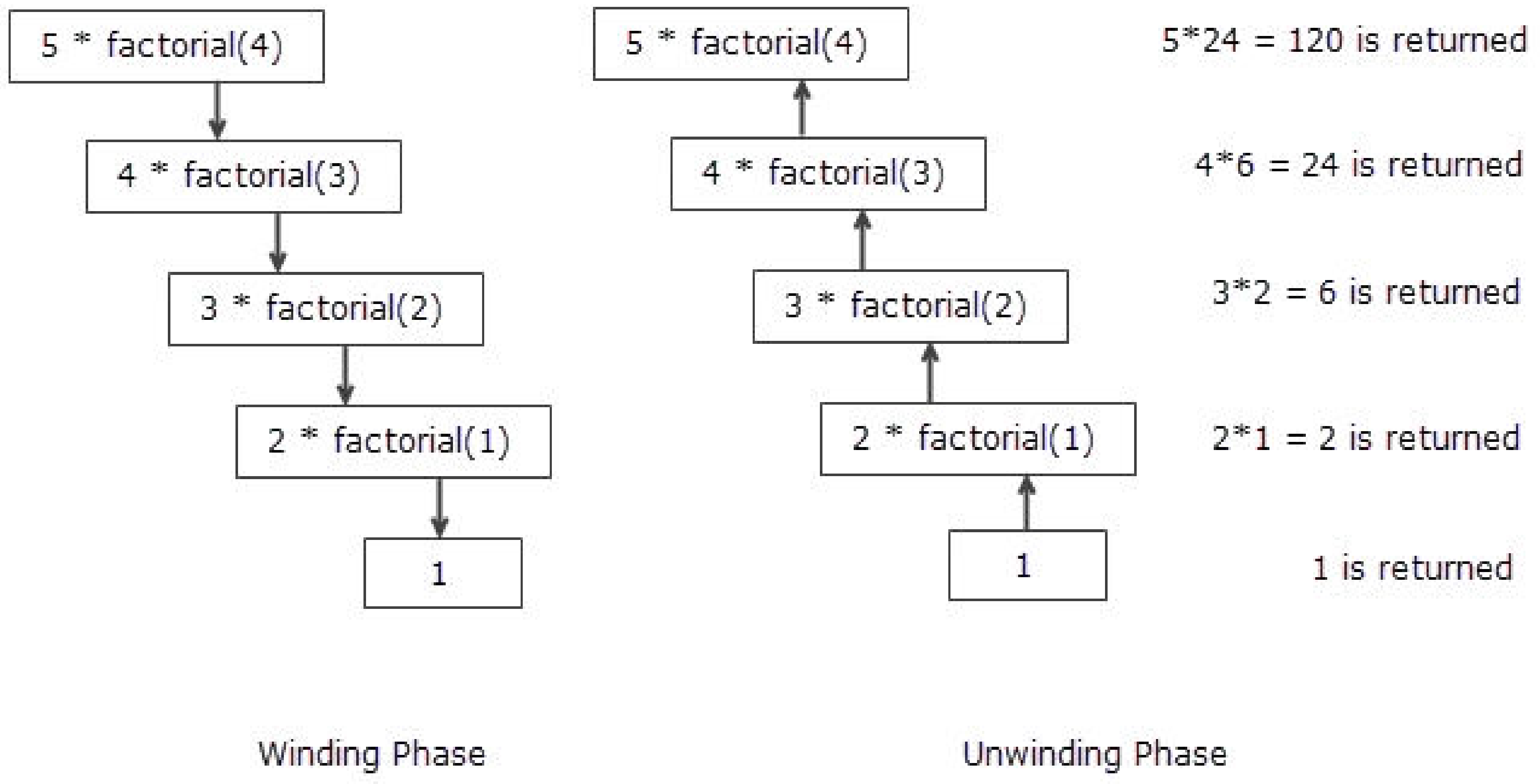
The recursive function works in two phases:

1. Winding phase.
2. Unwinding phase.

Winding phase: In Winding phase, the recursive function keeps calling itself. This phase ends when the base condition is reached.

Unwinding phase: When the base condition is reached, unwinding phase starts and control returns back to the original call.

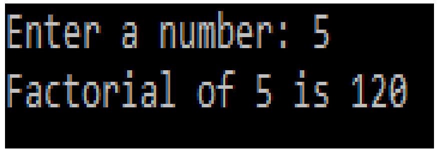
Recursion



5 * factorial(4)
4 * factorial(3)
3 * factorial(2)
2 * factorial(1)
1
Winding Phase
5 * factorial(4)
4 * factorial(3)
3 * factorial(2)
2 * factorial(1)
1
5*24 = 120 is returned
4*6 = 24 is returned
3*2 = 6 is returned
2*1 = 2 is returned
1 is returned
Unwinding Phase

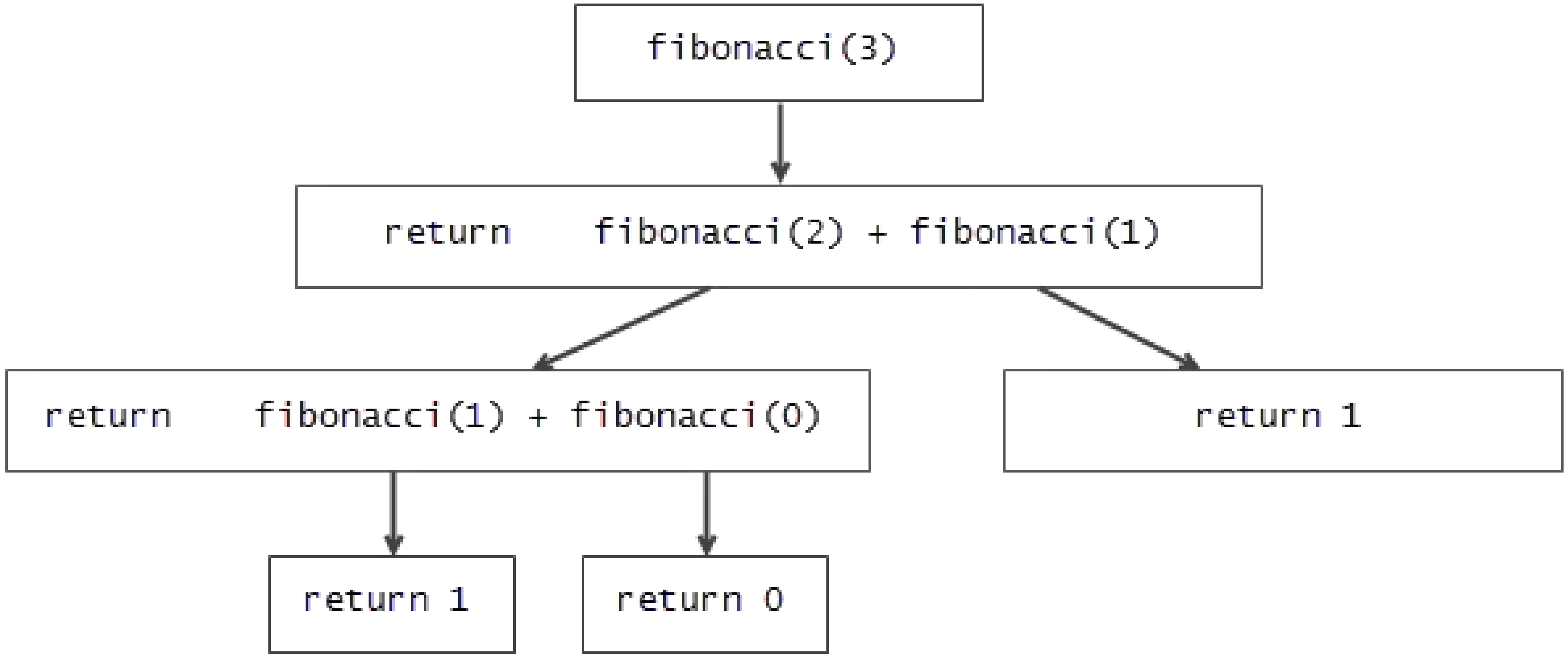
Recursion

```
//Factorial Using Recursion
#include<stdio.h>

long factorial(int n);

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

   fact = factorial(number);
   printf("Factorial of %d is %ld\n", number, fact);
}

long factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return(n * factorial(n-1));
}
```



```
Enter a number: 5
Factorial of 5 is 120
```

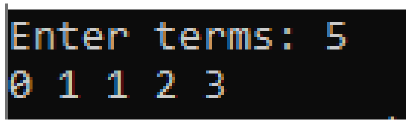
Recursion


fibonacci(3)
return fibonacci(2) + fibonacci(1)
return fibonacci(1) + fibonacci(0)
return 1
return 1
return 0

Evaluation of fibonacci(3)

Recursion

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

void main(void)
{
	int terms;
	printf("Enter terms: ");
	scanf("%d", &terms);

	for(int n = 0; n < terms; n++)
	{
		printf("%d ", fibonacci(n));
	}
}

int fibonacci(int num)
{
	//base condition
	if(num == 0 || num == 1)
	{
		return num;
	}
	else
	{
		// recursive call
		return fibonacci(num-1) + fibonacci(num-2);
	}
}
```



```
Enter terms: 5
0 1 1 2 3
```

Iteration VS Recursion

Example: Recursive Factorial Function

Iteration Definition:

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

Recursion Definition:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n=0 & \text{(Base Case)} \\ n*\text{fact}(n-1) & \text{if } n>0 & \text{(General Case)} \end{cases}$$

Iteration VS Recursion

Iteration	Recursion
A set of instructions repeatedly executed.	Function calls itself.
Application: loops	**Application:** functions.
Iteration is terminated when the loop condition fails	Recursion is terminated when base case is satisfied
Used when time complexity needs to be balanced against an expanded code size.	Used when code size needs to be small, and time complexity is not an issue.
Larger Code Size.	Smaller code size
Relatively lower time complexity(generally polynomial-logarithmic).	Very high(generally exponential) time complexity.
Execution is faster	Execution is slower
If condition is never false, it leads to infinite iteration with computers CPU cycle being used repeatedly.	If base case is never reached it leads to infinite recursion leading to memory crash.
memory usage is less.	memory usage is high.

Practical-9.3

A positive integer is entered through the keyboard, write a function to find the binary equivalent of this number using recursion.

Recursion

1. Write a program to find the reverse number of entered number using recursive function.
2. Write a program to calculates the power of a number using recursion.

Passing 1D Array to Functions

Like the values of simple variables, it is also possible to pass the values of an array to function.

Syntax:

Function call:

```
function-name(array-name,array-size);
```

Example:

```
largest(a,n);
```

Function Header:

```
float largest(float array[],int size)
```

NOTE: It is not necessary to specify the size of the array here.

Passing 1D Array to Functions

- In C, the **name of the array represents the address of its first element.**
- By passing the array name, we are **passing the address** of the array to the called function.
- The array in the called function now **refers to the same array stored in the memory**
- Therefore any **changes in the array in the called function will be reflected in the original array.**

Passing 1D Array to Functions

Three rules to pass an Array(1D) to a Function:

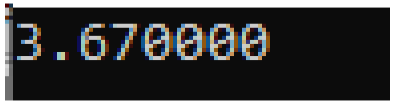
1. The function must be called by passing only the **name of the array**.

2. In the **function definition**, the formal parameter must be an array type; the size of the array does not need to be specified.

3. The **function prototype** must show that the argument is an array

Passing 1D Array to Functions

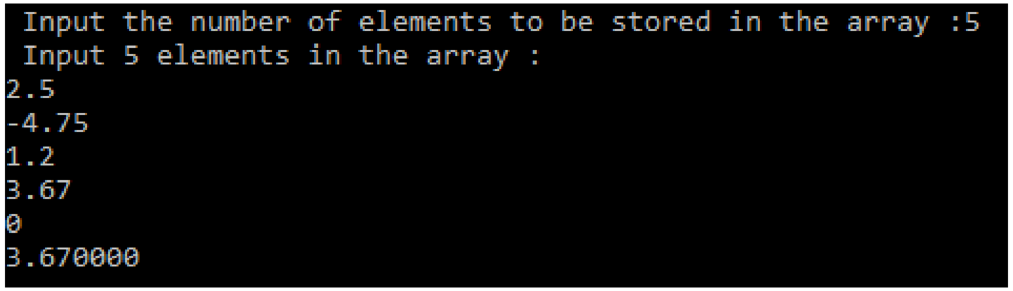
```
//Find the largest value in an array using function
float largest(float a[],int n);
main()
{
    float value[4]={2.5,-4.75,1.2,3.67};
    printf("%f\n",largest(value,4));
}
float largest(float a[],int n)
{
    int i;
    float max;
    max=a[0];
    for(i=1;i<n;i++)
        if(max<a[i])
            max=a[i];
    return max;
}
```



```
3.670000
```

Passing 1D Array to Functions

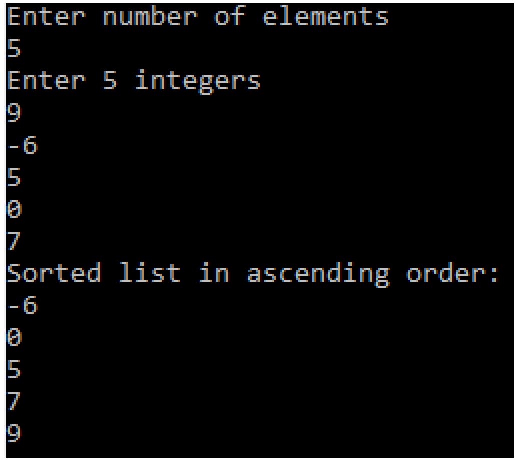
```c
//Find the largest value in an array using function
//Here we have not specify the size of array in argument
float largest(float a[]);
int n;

main()
{
    float value[10];
    int i;
    printf(" Input the number of elements to be stored in the array :");
    scanf("%d",&n);
    printf(" Input %d elements in the array :\n",n);
    for(i=0;i<n;i++)
        scanf("%f",&value[i]);
    printf("%f\n",largest(value));
}

float largest(float a[])
{
    int i;
    float max;
    max=a[0];
    for(i=1;i<n;i++)
        if(max<a[i])
            max=a[i];
    return max;
}
```



```
 Input the number of elements to be stored in the array :5
 Input 5 elements in the array :
2.5
-4.75
1.2
3.67
0
3.670000
```

Passing 1D Array to Functions

```
//Sort an array of integers using function
#include <stdio.h>
void sort (int m,int x[]);
void main()
{
   int c[20], n, i;
   printf("Enter number of elements\n");
   scanf("%d", &n);

   printf("Enter %d integers\n", n);
   for (i = 0; i < n; i++)
     scanf("%d", &c[i]);

     sort(n,c);

      printf("Sorted list in ascending order:\n");
   for (i = 0; i < n; i++)
      printf("%d\n", c[i]);
}
void sort (int m,int x[])
{
     int i,j,swap;

     for (i = 0 ; i < m-1; i++)
   {
     for (j = 0 ; j < m-1-i; j++)
     {
       if (x[j] > x[j+1])
       {
         swap    = x[j];
         x[j]    = x[j+1];
         x[j+1]  = swap;
       }
     }
   }
}
```



```
Enter number of elements
5
Enter 5 integers
9
-6
5
0
7
Sorted list in ascending order:
-6
0
5
7
9
```

Passing 1D Arrays to Functions

1. Write a program to find out the positions of the smallest and the largest elements of an array using function.
2. Write a Program to compute the standard Deviation of N Numbers using Arrays &
   Function. Use pow() and sqrt() function of maths.h header file.

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

Passing 2D Array to Functions

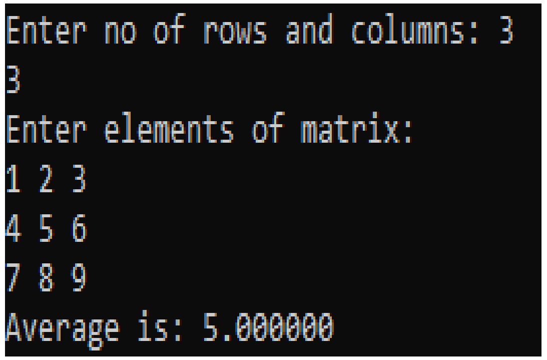
We can also pass multi-dimensional arrays to functions

Rules to pass an Array(2D) to a Function :

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets
3. The size of the second dimension must be specified
4. The prototype declaration should be similar to the function header.

Passing 2D Array to Functions

```c
#include <stdio.h>

float average(int x[][30],int row,int col)
{
	int i,j;
	int sum=0;

	for(i=0;i<row;i++)
	{
		for(j=0;j<col;j++)
		{
			sum+=x[i][j];
		}
	}
	return ((float)sum/(float)(row*col));
}

void main()
{
	int r,c,i,j,arr[30][30];
	float avg;
	printf("Enter no of rows and columns: ");
	scanf("%d %d",&r,&c);

	printf("Enter elements of matrix:");
	for(i=0;i<r;i++)
		for(j=0;j<c;j++)
			scanf("%d",&arr[i][j]);

	avg=average(arr,r,c);
	printf("Average is: %f\n",avg);
}
```



```
Enter no of rows and columns: 3
3
Enter elements of matrix:
1 2 3
4 5 6
7 8 9
Average is: 5.000000
```

Passing 2D Array to Functions

1. Write a user-defined function which multiplies two square matrices and also test this function
2. Write a user-defined function which add two square matrices and also test this function
3. Write a user-defined function for Transpose of a Matrix
4. Write a Program to find the upper triangle in the given matrix. Consider the following 4 x 4 Matrix.

$$\begin{matrix} X & X & X & X \\ 0 & X & X & X \\ 0 & 0 & X & X \\ 0 & 0 & 0 & X \end{matrix}$$

If all the elements denoted by X are non-zero, then the matrix has upper triangle. For the upper triangle, all the elements of principle diagonal and above must be non – zero. Pass two dimensional arrays to the function.

Passing 1D Strings to Functions

Strings are treated as character arrays in C

Rules to pass Strings to a Function

1. The string to be passed must be declared as a formal argument of the function when it is defined.

   For example:

```
void display (char item_name[ ] )
{
        .........
        .........
}
```

2. The function prototype must show that the argument is a string. For example `void display (char str[ ] );`

Passing 1D Strings to Functions

3. A call to the function must have a string array name without subscripts as its actual arguments.

Example:

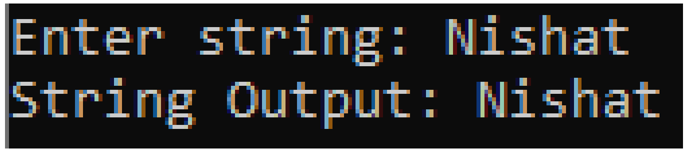
```
display(names);
```

Where names is a properly declared string array in the calling function.

Note: Both arrays and string cannot be passed by values to function.

Passing 1D Strings to Functions

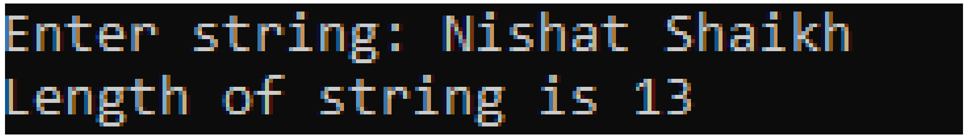
```c
#include <stdio.h>
void displayString(char str[]);

void main()
{
    char str[50];
    printf("Enter string: ");
    gets(str);
    displayString(str);
}

void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```



```
Enter string: Nishat
String Output: Nishat
```

Passing 1D Strings to Functions

```c
//Find length of string
#include <stdio.h>
int getLength(char str[]);

void main()
{
    char str[50];
    printf("Enter string: ");
    gets(str);
    printf("Length of string is %d",getLength(str));

}

int getLength(char str[])
{
    int i,length=0;
    for(i=0;str[i]!=0;i++)
        length++;
    return length;
}
```

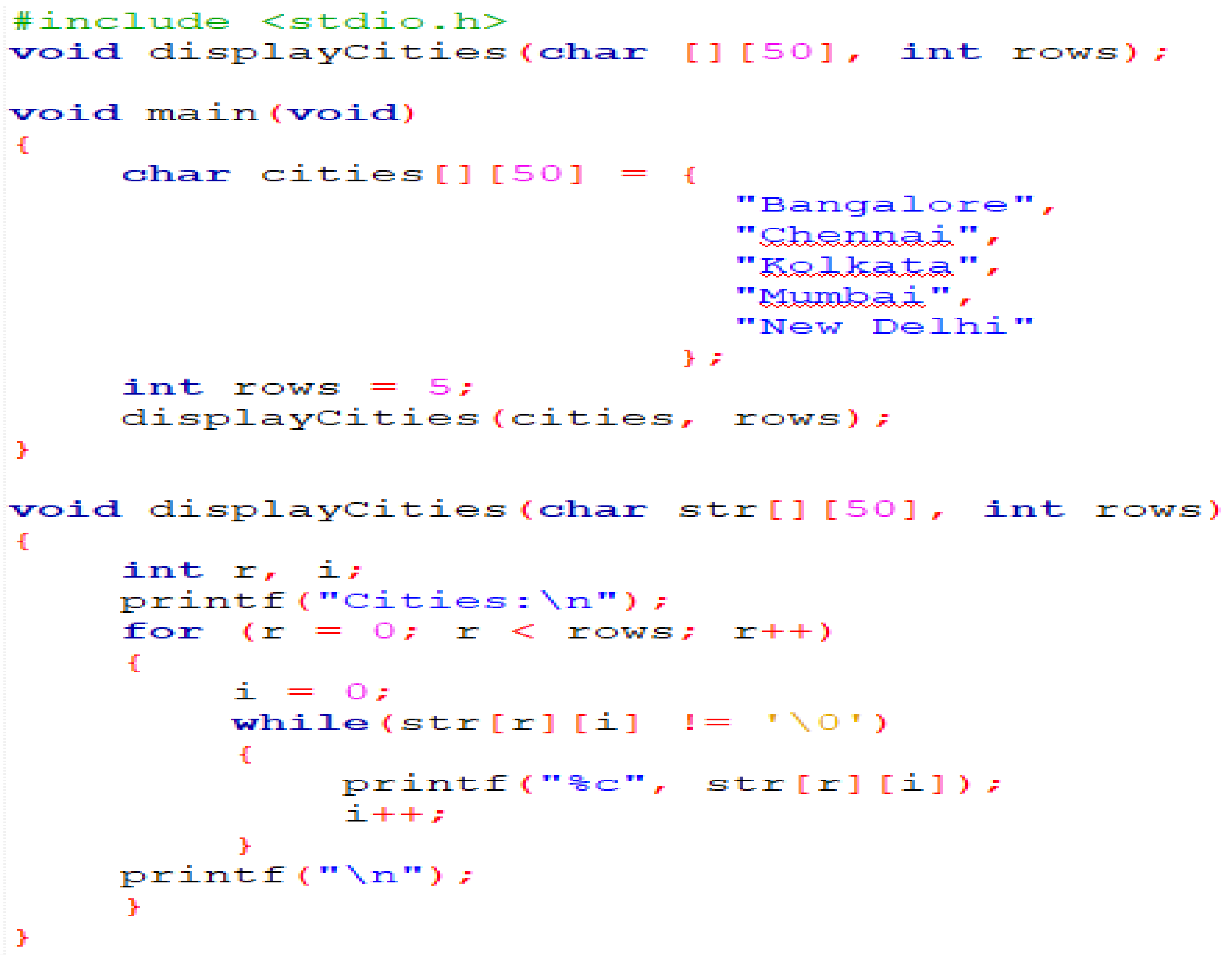


```
Enter string: Nishat Shaikh
Length of string is 13
```

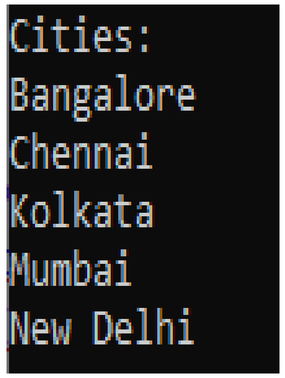
Passing 1D Strings to Functions

1. Write a Program to reverse a string using Recursive Function and check whether it is palindrome or not.

Passing 2D Strings to Functions



```c
#include <stdio.h>
void displayCities(char [][50], int rows);

void main(void)
{
	char cities[][50] = {
				"Bangalore",
				"Chennai",
				"Kolkata",
				"Mumbai",
				"New Delhi"
			    };
	int rows = 5;
	displayCities(cities, rows);
}

void displayCities(char str[][50], int rows)
{
	int r, i;
	printf("Cities:\n");
	for (r = 0; r < rows; r++)
	{
		i = 0;
		while(str[r][i] != '\0')
		{
			printf("%c", str[r][i]);
			i++;
		}
	printf("\n");
	}
}
```



```
Cities:
Bangalore
Chennai
Kolkata
Mumbai
New Delhi
```

Passing 2D Strings to Functions

1. Write a user-defined function which will take two-dimensional array of words and arrange them in dictionary order and also test this function

Scope, Visibility and Lifetime of Variables

Scope:

The region of a program in which a variable is **available** for use(**active**).

Visibility:

The program's ability to **access** a variable from the memory

Lifetime(longevity):

The lifetime of a variable is the duration of time in which a variable **exists** in the memory during execution(**alive**)

Types of Scope

1. **Program Scope** - Variable is available throughout the program.
2. **File Scope** - Variable is available throughout the file.
3. **Block scope** - Variable is available throughout the block in which it is declared.
4. **Function Scope** - Variable is available throughout the function in which it is declared.(usually for labels)
5. **Prototype Scope** - Variables passes as parameters to a function have this type of scope.

NOTE: Visibility is a subset of Scope. A variable may be available throughout the program, but it may not be accessible at some parts.

Storage Class

In C A variable is not only associated with a data type but also a storage class.

Storage Classes are used to describe the features of a variable/function. These features basically include:

- The variable **scope**.
- Who can access a variable?(**visibility)**
- A **lifetime** of a variable
- The location where the variable will be stored**(Memory/ CPU Registers)**
- The initial value of a variable(**Default Value**)

Storage Class

There are FOUR types of storage classes

- Automatic Storage class (auto)
- Register Storage class (register)
- Static Storage class (static)
- External Storage class (extern)

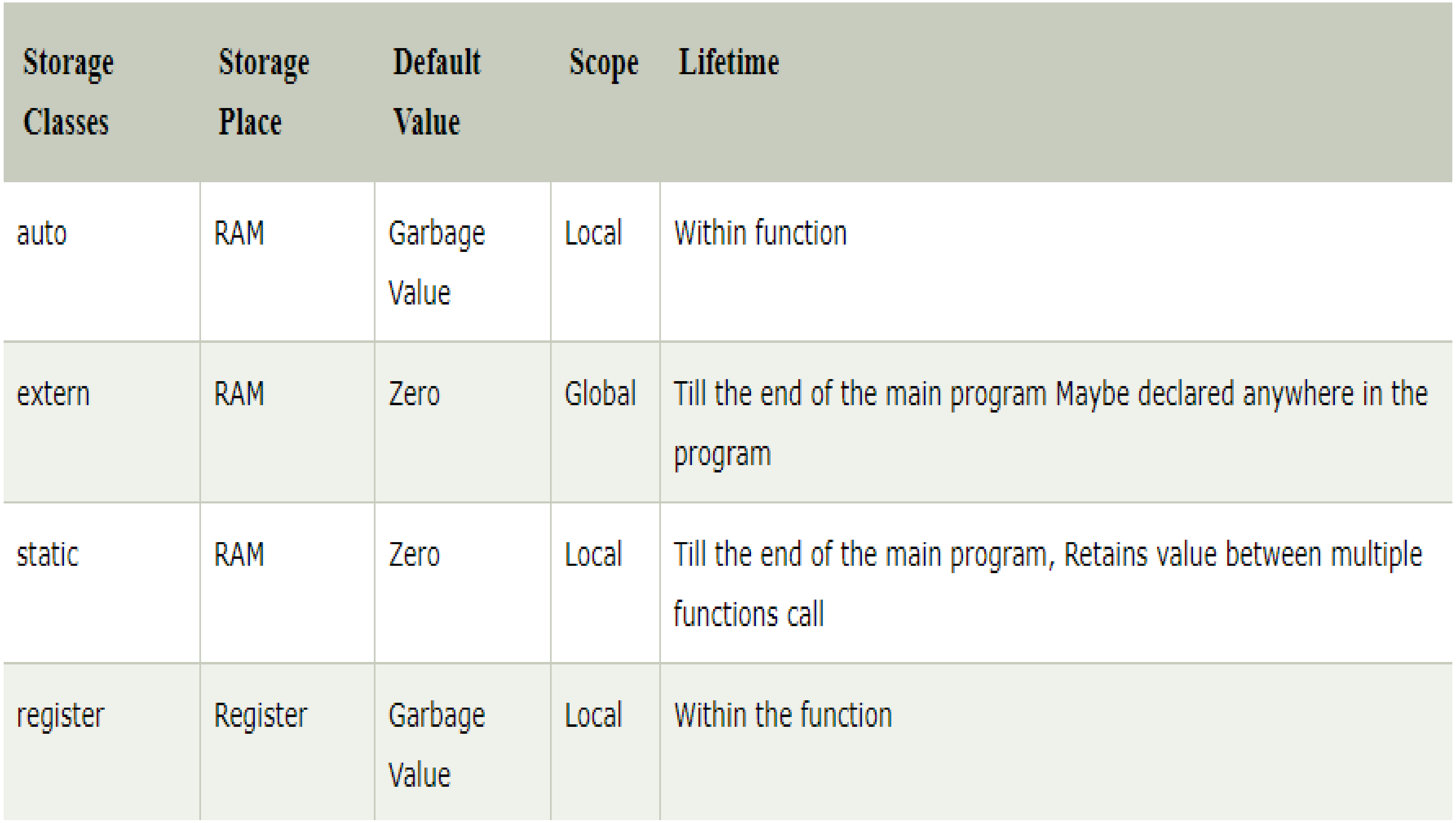
Syntax:

```
storage_class  var_data_type  var_name;
```

Example:

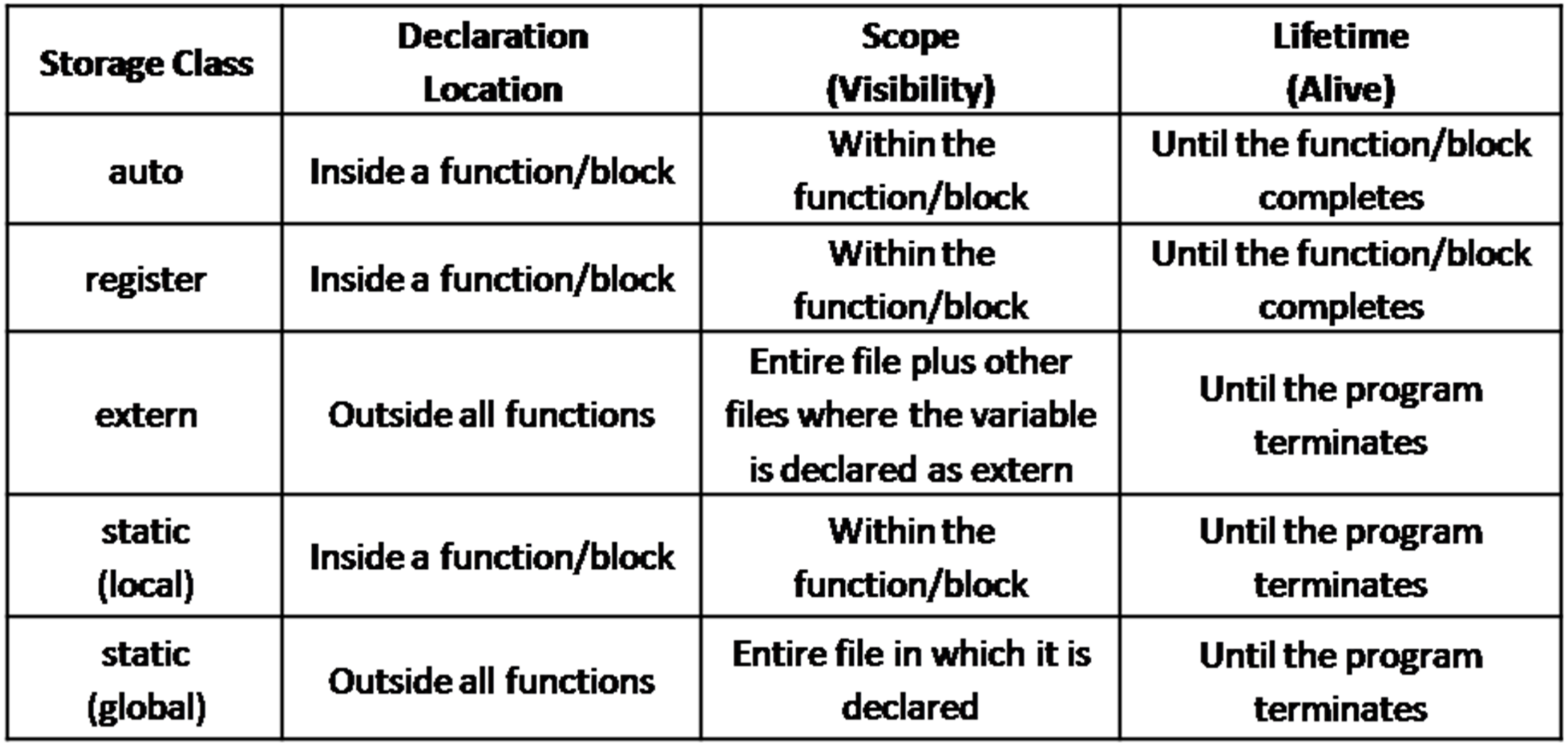
```
auto int a;
extern int a;
static int a;
register int a;
```

Storage Class



Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

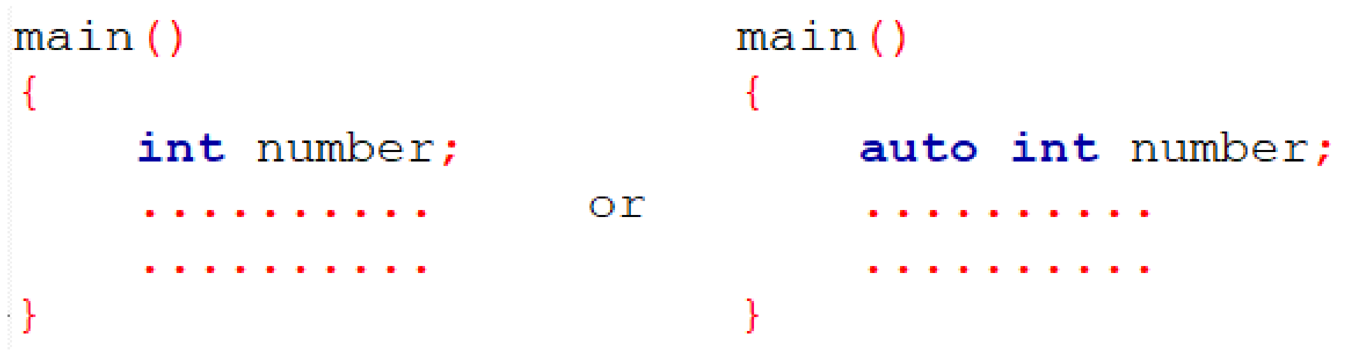
Scope and Lifetime of variables



Storage Class	Declaration Location	Scope (Visibility)	Lifetime (Alive)
auto	Inside a function/block	Within the function/block	Until the function/block completes
register	Inside a function/block	Within the function/block	Until the function/block completes
extern	Outside all functions	Entire file plus other files where the variable is declared as extern	Until the program terminates
static (local)	Inside a function/block	Within the function/block	Until the program terminates
static (global)	Outside all functions	Entire file in which it is declared	Until the program terminates

Automatic variables

- Declared **inside a function**
- **Created** when the function is called
- **destroyed** automatically when the function is exited, hence the name **automatic**
- Also referred to as **local or internal variables** as they are private/local to the function in which they are declared
- A variable declared inside a function without storage class specification is, **by default, an automatic variable**.



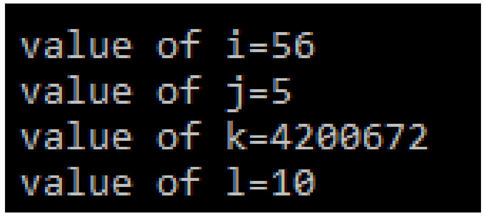
```
main()
{
     int number;
     ...........
     ...........
}
```

or

```
main()
{
     auto int number;
     ...........
     ...........
}
```

Automatic variables

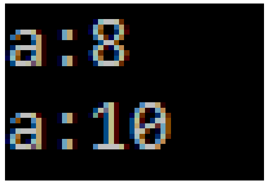
```
//Demo of auto with default value
main()
{
    auto int i,j=5;
    int k,l=10;
    printf("\n value of i=%d\n value of j=%d",i,j);
    printf("\n value of k=%d\n value of l=%d",k,l);
}
```



```
value of i=56
value of j=5
value of k=4200672
value of l=10
```

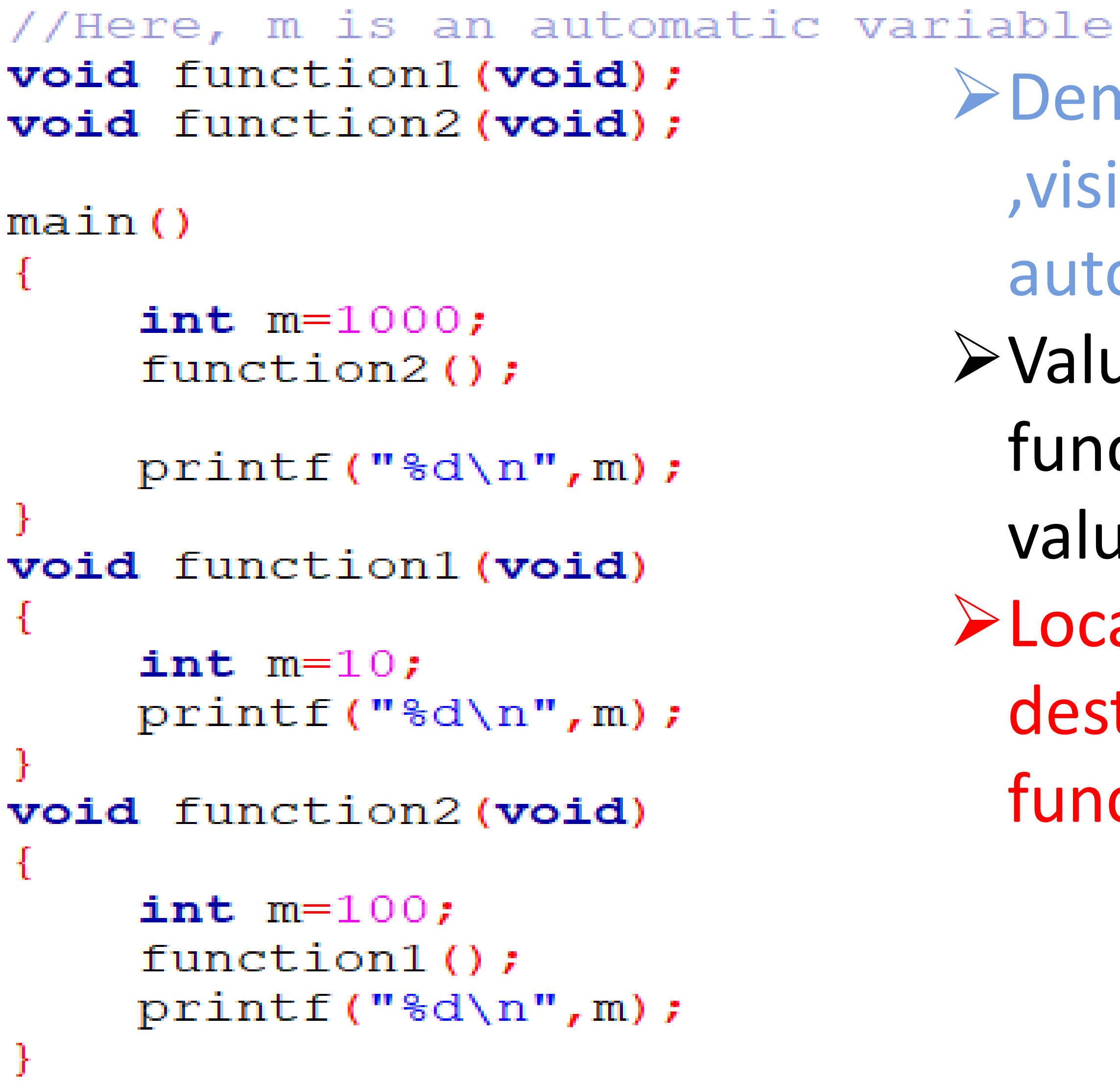
Automatic variables

```
//auto int a; //Error
void main()
{
    auto int a=10;  //Function scope
    {
        auto int a;  //Block Scope
        printf("a:%d\n",a);  //Garbage Value
    }
    printf("a:%d\n",a); //10
}
```



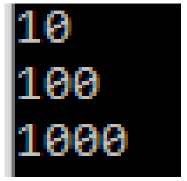
```
a:8
a:10
```

Automatic variables



```c
//Here, m is an automatic variable
void function1(void);
void function2(void);

main()
{
     int m=1000;
     function2();

     printf("%d\n",m);
}
void function1(void)
{
     int m=10;
     printf("%d\n",m);
}
void function2(void)
{
     int m=100;
     function1();
     printf("%d\n",m);
}
```

- Demonstration of scope ,visibility and lifetime of auto storage class.
- Value assigned to m in one function does not affect its value in the other function
- Local value of m is destroyed when it leaves a function



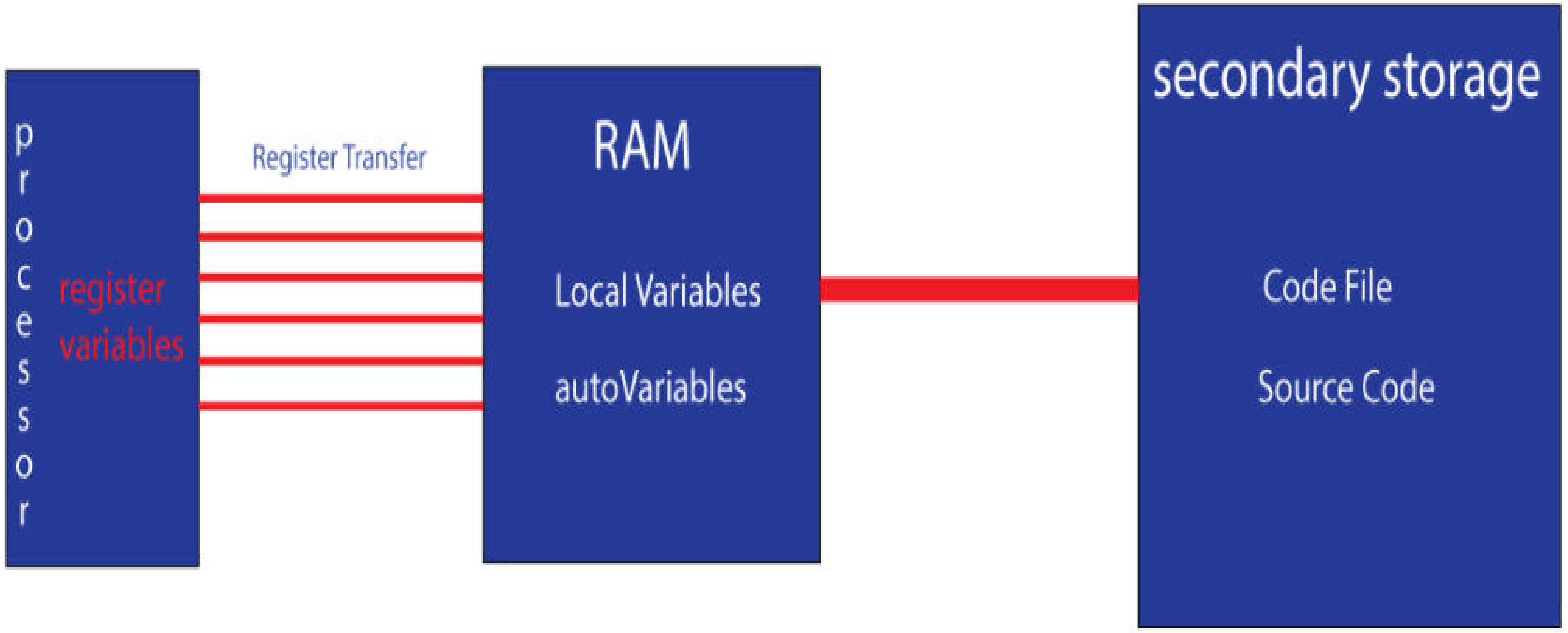
```
10
100
1000
```

Register variables

- We can tell the compiler that a variable should be kept in one of the **machine's register**, instead of keeping in the memory (where normal variables are stored)
- Since a register access is much faster than a memory access, keeping the frequently accessed variables(e.g., loop control variables) in the register will lead to **faster execution of programs**.

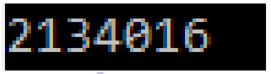
```
register int count;
```

Register variables



Register variables

```c
// The initial default value of a is Garbage
#include <stdio.h>
int main()
{
register int a; //a is allocated memory in the CPU register
printf("%d",a);
}
```

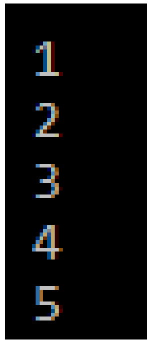


```
2134016
```

```c
//compile time error since we cannot access the address of a register variable
#include <stdio.h>
int main()
{
    register int a = 0;
    printf("%u",&a);
}
```

error: address of register variable 'a' requested

```c
//Speed of the program increases by using a variable as register storage clas
void main()
{
    register  int  i;
    for(i=1;i<=5;i++)
    {
        printf("\n %d",i);
    }
}
```



```

 1
 2
 3
 4
 5
```

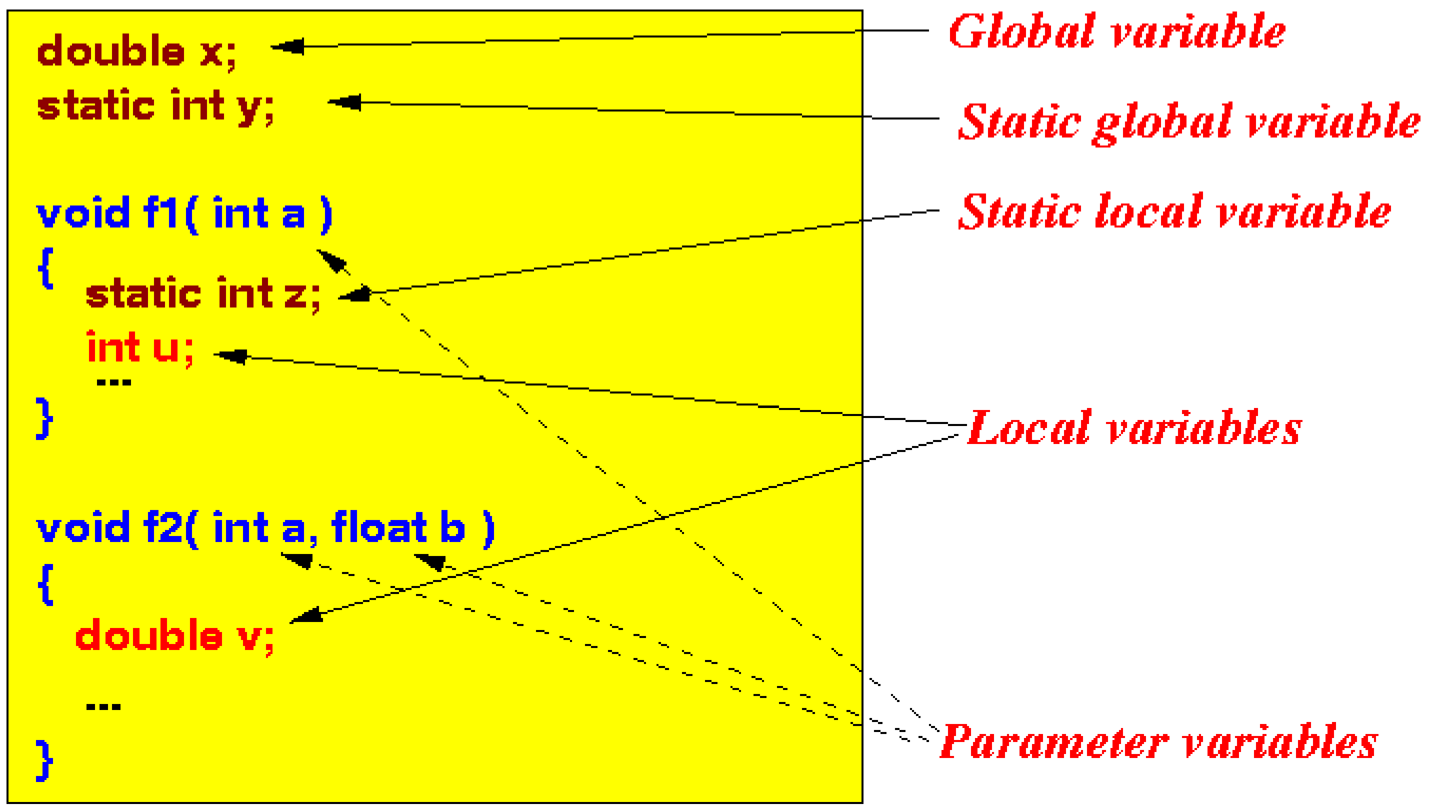
static variables

- The value of static variables persists until the end of the program.

```
static int x;
```

- Depending on the place of declaration, static variable can be
  - **Internal static variable**
  - **External static variable**

static variables



```
double x;
static int y;

void f1( int a )
{
   static int z;
   int u;
   ...
}

void f2( int a, float b )
{
   double v;

   ...
}
```

static variables

Internal static variable

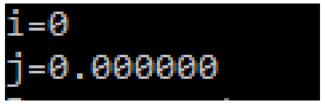
- Declared inside function
- Scope: Only in that function
- **Similar to auto variables (except lifetime**)
  - Lifetime(auto):within function/block
  - Lifetime(internal static):Global

External static variable

- Declared outside all function
- **Difference between static external variable and simple external variable** is that static external variable is available only within the file where it is defined while simple external variable can be accessed by other files

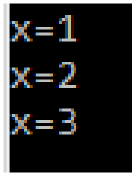
static variables

```c
// initial default value of static variable
#include<stdio.h>
static int i;
void main ()
{
     static float j;
     printf("i=%d\nj=%f",i,j);
}
```



```
i=0
j=0.000000
```

```c
// static variable is initialized only once
void stat(void);
void main()
{
     int i;
     for(i=1;i<=3;i++)
          stat();
}
void stat(void)
{
     static int x=0;
     x=x+1;
     printf("x=%d\n",x);
}
```



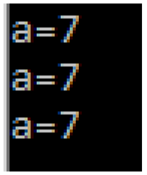
```
x=1
x=2
x=3
```

NOTE: replace static with auto and observe the outpu

Auto Vs Static Local variables

auto

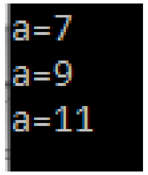
```
void fun();

void main()
{
    fun();
    fun();
    fun();
}

void fun()
{
    auto int a=5;
    printf("a=%d\n",a+=2);
}
```



```
a=7
a=7
a=7
```

Static local

```
void fun();

void main()
{
    fun();
    fun();
    fun();
}

void fun()
{
    static int a=5;
    printf("a=%d\n",a+=2);
}
```



```
a=7
a=9
a=11
```

External variables

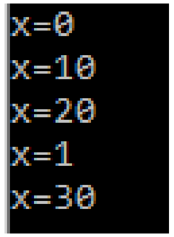
- Variables that are both **alive and active** throughout the program are known as external variables
- Also known as **global variables**
- Can be accessed by any function in the program
- Declared **outside a function**

```
int number;                             extern int number;
void main()                             void main()
{                                       {
      .......                                 .......
      .......                                 .......
}                                       }
void f1()                               void f1()
{                    OR                 {
      .......                                 .......
      .......                                 .......
}                                       }
void f2()                               void f2()
{                                       {
      .......                                 .......
      .......                                 .......
}                                       }
```

External variables

```
int  f1();
int  f2();
int  f3();

int  x;  //GLOBAL

void main()
{
     printf("x=%d\n",x);
     x=10;
     printf("x=%d\n",x);
     printf("x=%d\n", f1());
     printf("x=%d\n", f2());
     printf("x=%d\n", f3());
}

int  f1()
{
     x=x+10;
}
int  f2()
{
     int  x;  //LOCAL
     x=1;
     return  x;
}
f3()   //return_type by default int
{
     x=x+10;  //GLOBAL
}
```

NOTE: when local and global variable have same name, the local variable will have precedence over the global one



```
x=0
x=10
x=20
x=1
x=30
```

External variables

Problem

```c
void main()
{
    y=5;  //ERROR:y undeclared
    f1();
}

int y;

int f1()
{
    y=y+1;  //assign 1 to y
}
```

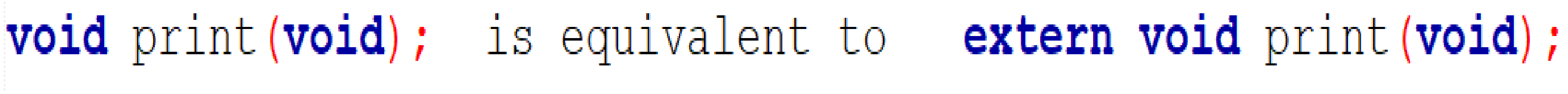
Solution

```c
void main()
{
    extern int y;  //external declaration
    ........
}

int f1()
{
    extern int y;  //external declaration
    ........
}

int y;  //definition
```

External variables

NOTE:

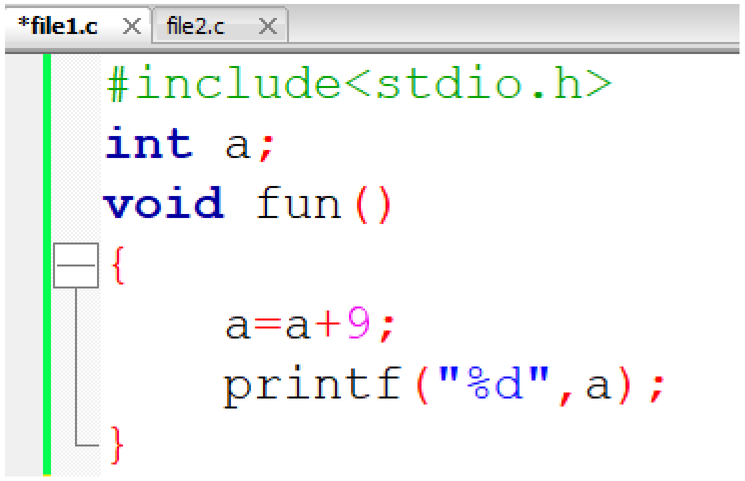
- The external declaration of y inside the function informs the compiler that y is an integer type **defined somewhere else in the program.**
- Extern declaration **does not allocate storage space** for variables
- In case of arrays, the definition should include their size as well.



```
void print(void);   is equivalent to   extern void print(void);
```

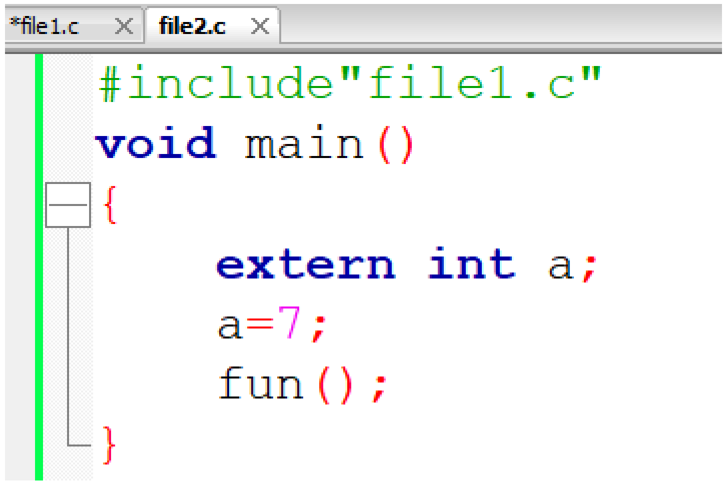
External variables

file1.c

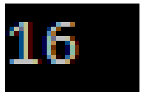


```c
#include<stdio.h>
int a;
void fun()
{
    a=a+9;
    printf("%d",a);
}
```

file2.c



```c
#include"file1.c"
void main()
{
    extern int a;
    a=7;
    fun();
}
```



```
16
```

End of Unit-09