UNIT 5

- **Functions** – Function prototype, function return type, signature of a function, function arguments, call by value, Function call stack and Activation Records, Recursion v/s Iteration, passing arrays (single and multi-dimensional) to functions,
- **Storage classes-** Automatic, Static, Register, External, Static and Dynamic linking implementation, C program memory (show different areas of C program memory and where different type of variables are stored), scope rules

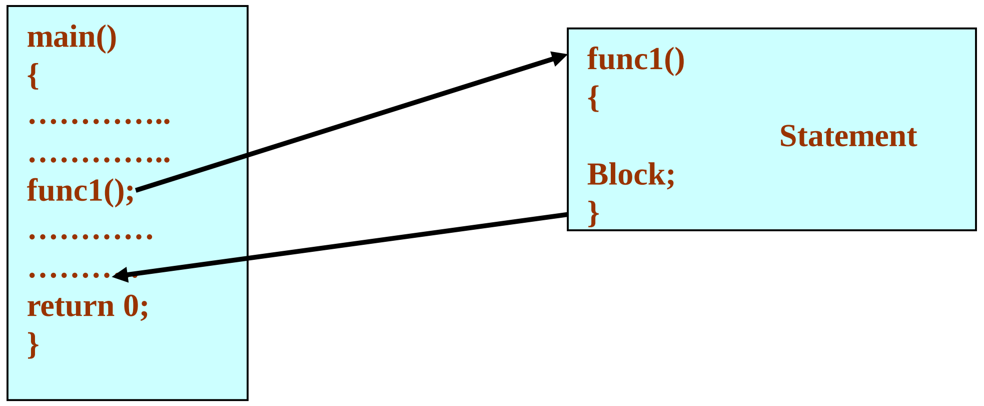
INTRODUCTION (Functions)

A function is a self-contained subprogram that is meant to do some specific, well-defined task.

If a program has only one function then it must be the main() function.

- Every function in the program is supposed to perform a well defined task. Therefore, the program code of one function is completely insulated from that of other functions.
- Every function has a name which acts as an interface to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it.

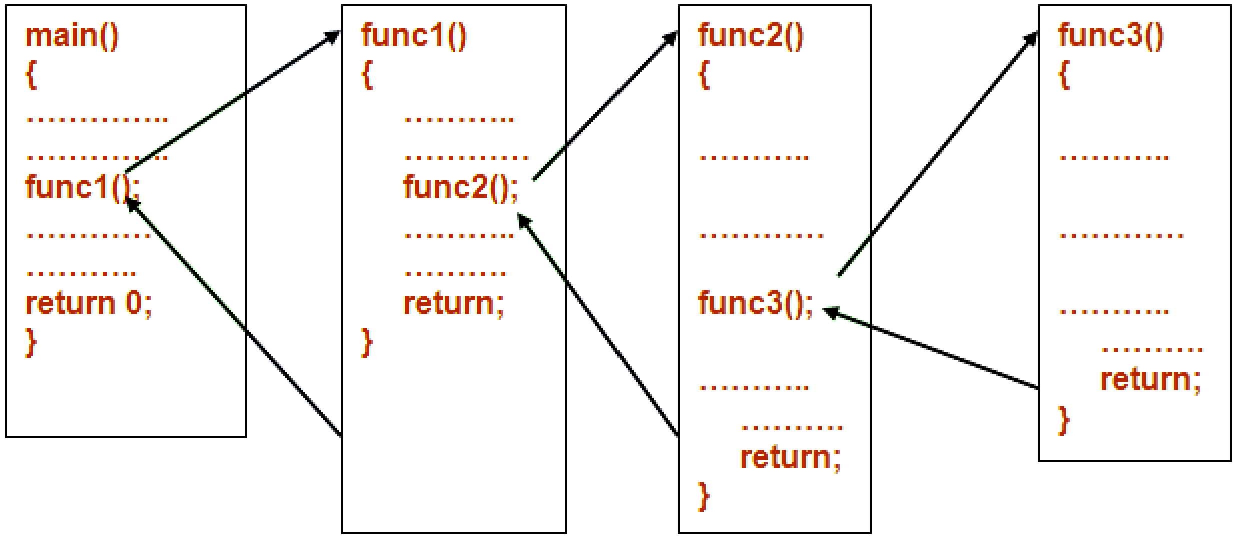
- In the fig, main() calls another function, func1() to perform a well defined task.
- main() is known as the calling function and func1() is known as the called function.



- When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function.
- After the called function is executed, the control is returned back to the calling program.

- A function, ***f*** that uses another function ***g***, is known as the ***calling function*** and g is known as the ***called function***.
- The inputs that the function takes are known as *arguments*.
- When a called function returns some result back to the calling function, it is said to *return* that result.
- *The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.*
- Main() is the function that is called by the operating system and therefore, it is supposed to return the result of its processing to the operating system.

- It is not necessary that the main() can call only one function, it can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for loop, while loop or do-while loop may call the same function multiple times until the condition holds true.
- It is not that only the main() can call another functions. Any function can call any other function. In the fig. one function calls another, and the other function in turn calls some other function.



WHY DO WE NEED FUNCTIONS?
Advantages Of Using Functions:

- Generally a difficult problem is divided into sub problems and then solved.
- This divide and conquer technique is implemented in C through functions. A program can be divided into functions, each of which performs some specific task.
- When some specific code is to be used more than once and at different places in the program the use of functions avoids repetition of that code.
- The program becomes easily understandable, modifiable and easy to debug and test. It becomes simple to write the program and understand what work is done by each part of the program.
- Functions can be stored in a library and reusability can be achieved.

C programs have two types of functions-
1. Library functions
2. User-defined functions

Library Functions

C has the facility to provide library functions for performing some operations. These functions are present in the C library and they are predefined.

For example **sqrt( )** is a mathematical library function which is used for finding out the square root of any number. The functions scanf( ) and printf( ) are input-output library functions. Similarly we have functions like strlen( ), strcmp( ) for string manipulation.

To use a library function we have to include corresponding header file using the preprocessor directive **#include**.

For example:

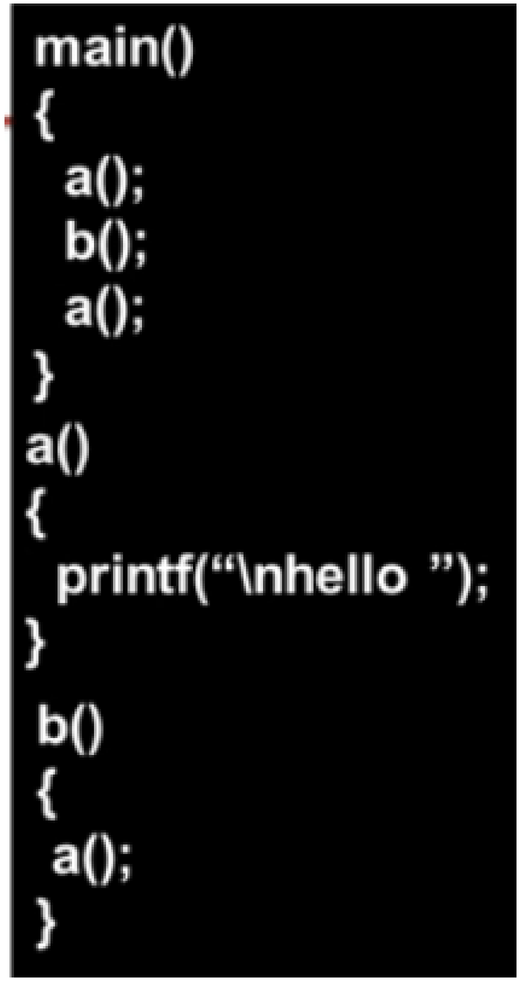
- to use input output functions like printf( ), scanf( ) we have to include **stdio.h**
- for mathematical library functions we have to include **math.h**
- for string library **string.h** should included.

User-Defined Functions

Users can create their own functions for performing any specific task of the program. These types of functions are called user-defined functions.

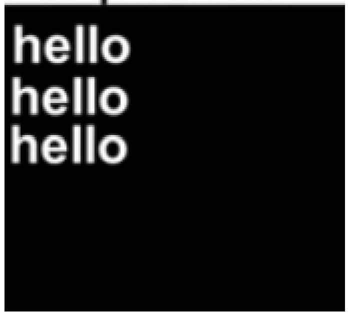
To create and use these functions, we should know about these three things-

1. **Function definition .**
2. **Function declaration**
3. **Function call**

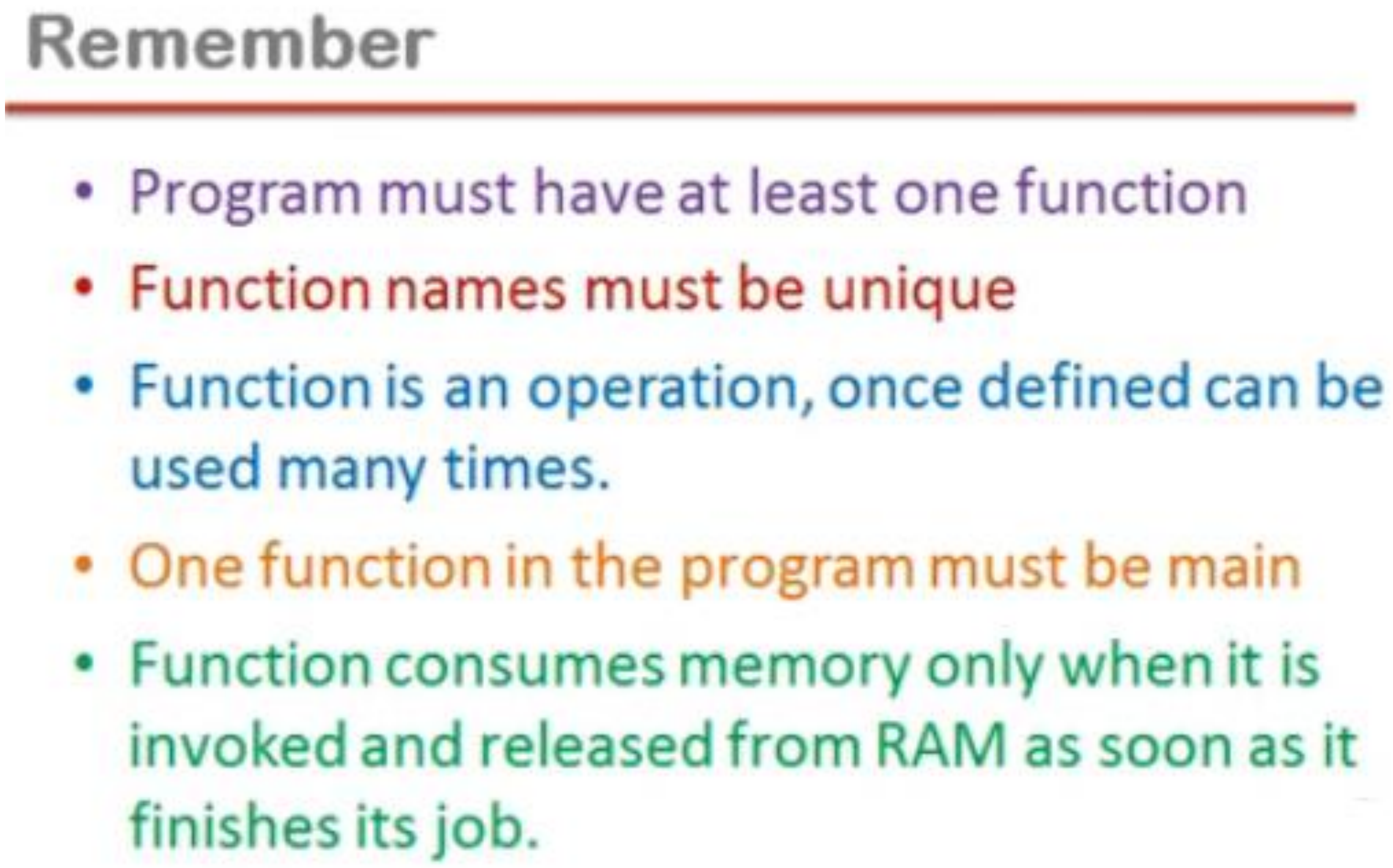


```
main()
{
  a();
  b();
  a();
}
a()
{
  printf("\nhello ");
}
b()
{
  a();
}
```

Output Screen

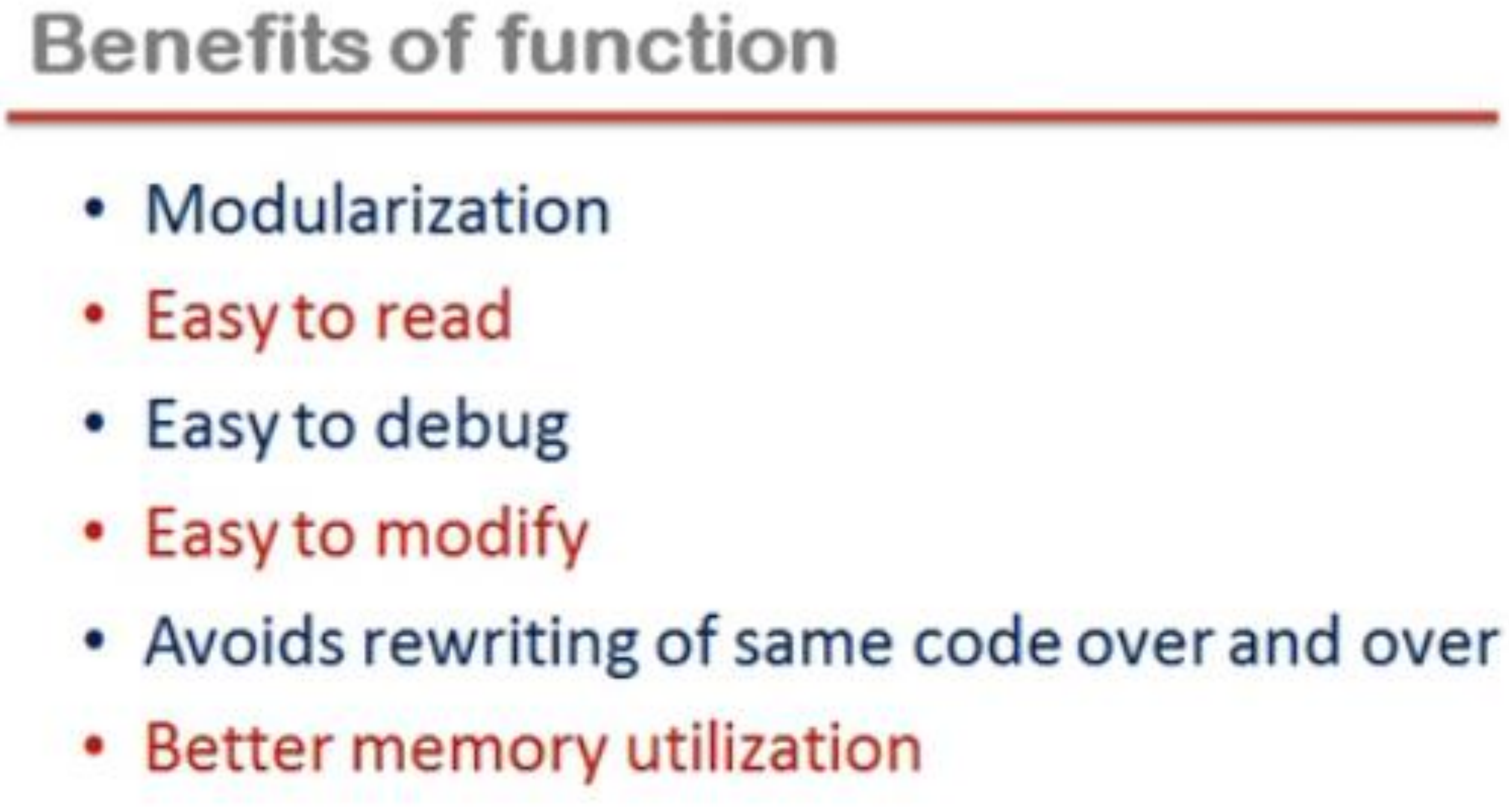


```
hello
hello
hello
```



Remember

- Program must have at least one function
- Function names must be unique
- Function is an operation, once defined can be used many times.
- One function in the program must be main
- Function consumes memory only when it is invoked and released from RAM as soon as it finishes its job.



Benefits of function

- Modularization
- Easy to read
- Easy to debug
- Easy to modify
- Avoids rewriting of same code over and over
- Better memory utilization

FUNCTION DECLARATION

- *Function declaration* is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.
- The function declaration is also known as the function prototype, and it informs the compiler about the following three things-
  1. Name of the function.
  2. Number and types of arguments received by the function.
  3. Type of value returned by the function.
- The general format for declaring a function that accepts some arguments and returns some value as result.

  return_type function_name (datatype variable1, datatype variable2,..);

  Example:

```
void drawline(void);
float avg ( int a, int b);
int sum(int x,  int y);
```

/ *Program to draw a line* /

```
#include<stdio.h>
void drawline(void);                    /*Function Declaration*/
main( )
{
drawline( );                            /*Function Call*/
}

void drawline(void)                     /*Function Definition*/
{
int i;
for(i=1;i<=80;i++)
printf("_" );
}
```

/*Program to find the sum of two numbers* /

```
#include<stdio.h>
int sum(int x, int y);              /*Function declaration*/
main( )
{
int a,b,total;
printf ("Enter values for a and b: ");
scanf ("%d%d",&a,&b);

total=sum (a, b);                   /*Function call*/

printf ("Sum of %d and %d is %d \n" ,a, b, total);
}

int sum(int x, int y)                              /*Function definition*/
{
int s;
s=x+y;
return s;
}
```

```
#include<stdio.h>
int sum(int a, int b);                    // FUNCTION DECLARATION
int main()
{
    int num1, num2, total = 0;
    printf("\n Enter the first number : ");
    scanf("%d", &num1);
    printf("\n Enter the second number : ");
    scanf("%d", &num2);

    total = sum(num1, num2);    // FUNCTION CALL
    printf("\n Total = %d", total);
    return 0;
}
                                          // FUNCTION DEFNITION
int sum ( int a, int b)                   // FUNCTION HEADER
{                                         //  FUNCTION BODY
    return (a + b);
}
```

Function Definition

- The function definition consists of the whole description and code of a function. It tells what the function is doing and what are its inputs and outputs.
- When a function defined, space is allocated for that function in the memory.
- A function definition consists of two parts
  - a function header and
  - a function body.
- The return type denotes the type of the value that will be returned by the function.
- The return type is optional and if omitted, it is assumed to be **int** by default.
- A function can return either one value or no value. (If a function does not return any value then void should be written in place of return type.)

Function Definition (continued..)

- ✓ After function name the argument declarations are given in parentheses, which mention the type and name of the arguments.
- ✓ These are known as formal arguments, and used to accept values.
- ✓ A function can take any number of arguments or even no argument at all.
- ✓ If there are no arguments then either the parentheses can be left empty or void can be written inside the parentheses.
- ✓ The variables declared inside. the function are known as local variables, since they are local to that function only, i.e. they have existence only in the function in which they are declared, they can not be used anywhere else in the program.
- ✓ The return statement is optional. It may be absent if the function does not return any value.
- ✓ The function definition can be placed anywhere in the program. But generally all definitions are placed after the main( ) function. Function definitions can also be placed in different files.

```
//Example 1: (considering the previous example)
    void drawline(void)                    /*Function Definition*/
    {
    int i;
    for(i=1;i<=80;i++)
    printf("_" );
    }
```

- Here the function is not returning any value so void is written at the place of return type, and since it does not accept any arguments so void is written inside parentheses.

- The int variable i is declared inside the function body so it is a local variable and can be used inside this function only.

//Example 2: (considering the previous example)

```
int   sum(int  x,    int  y)                         /*Function definition*/
{
int  s;
s = x + y;
return s;
}
```

- This function returns a value of type int ,because int is written at the place of return type.
- This function takes two formal arguments x and y, both of type int.
- The variables is declared inside the function, so it is a local variable.
- The formal arguments x and y are also used as local variables inside this function.

Function Call

- The function call statement invokes the function.
- When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.
- Function call statement has the following syntax.

function_name(variable1, variable2, …);

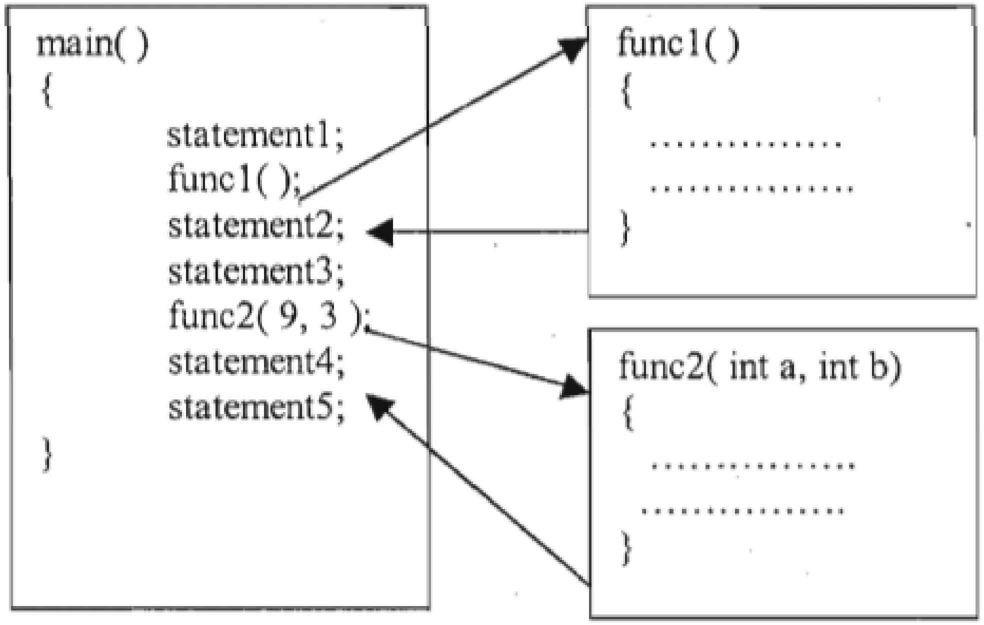
Points to remember while calling the function:

- Function name and the number and type of arguments in the function call must be same as that given in the function declaration and function header of the function definition.

•Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.

•If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.

variable_name = function_name(variable1, variable2, …);



Transfer of control when function is called

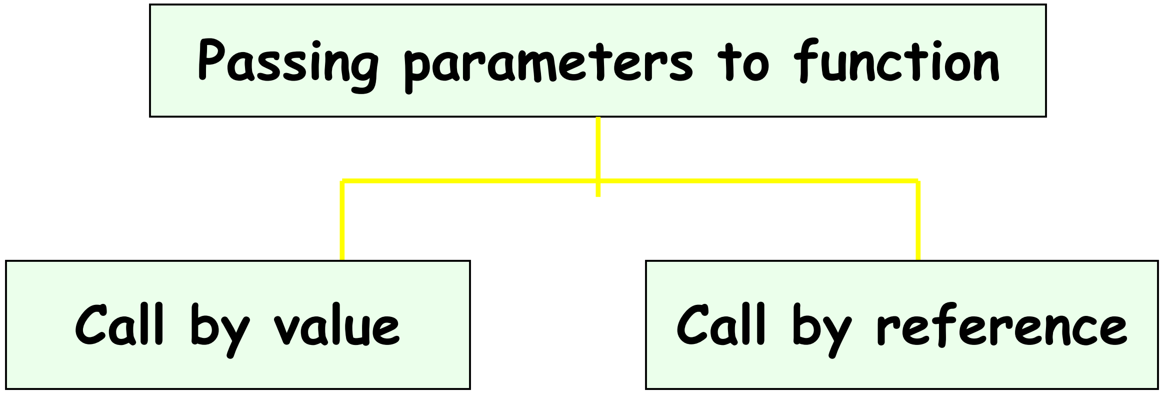
return statement

- The **return** statement is used to terminate the execution of a function and return control to the calling function. When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- **Programming Tip:** It is an error to use a return statement in a function that has void as its return type.
- Programmer may or may not place the *expression* within parentheses.
- By default, the return type of a function is *int.*
- For functions that has no **return** statement, the control automatically returns to the calling function after the last statement of the called function

  is executed.

PASSING PARAMETERS TO THE FUNCTION

- There are two ways in which arguments or parameters can be passed to the called function.



- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.

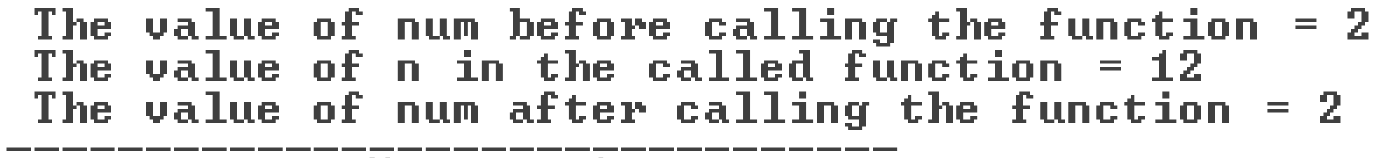
CALL BY VALUE

- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

// program to demonstrate cal by value.

```
#include<stdio.h>
void add( int n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
void add(int n)
{
    n = n + 10;
    printf("\n The value of n in the called function = %d", n);
}
```

//Output:



```
The value of num before calling the function = 2
The value of n in the called function = 12
The value of num after calling the function = 2
--------------------------------
```

CALL BY REFERENCE

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it received are visible by the calling program.
- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

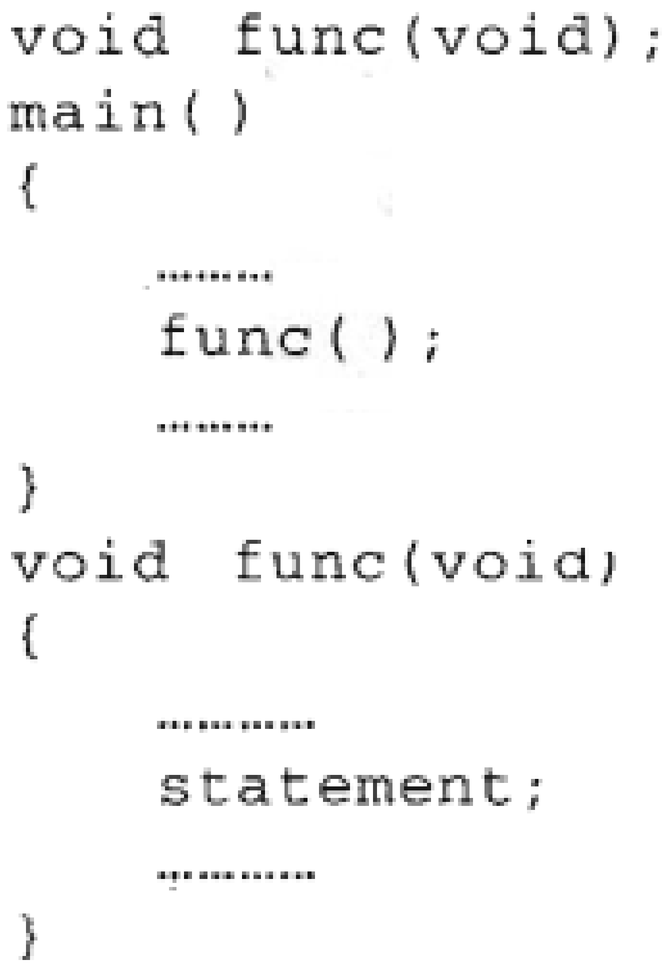
Types Of Functions

The functions can be classified into four categories on the basis of the arguments and return value.

1. **Functions with no arguments and no return value.**
2. **Functions with no arguments and a return value.**
3. **Functions with arguments and no return value.**
4. **Functions with arguments and a return value.**

Functions With No Arguments And No Return Value

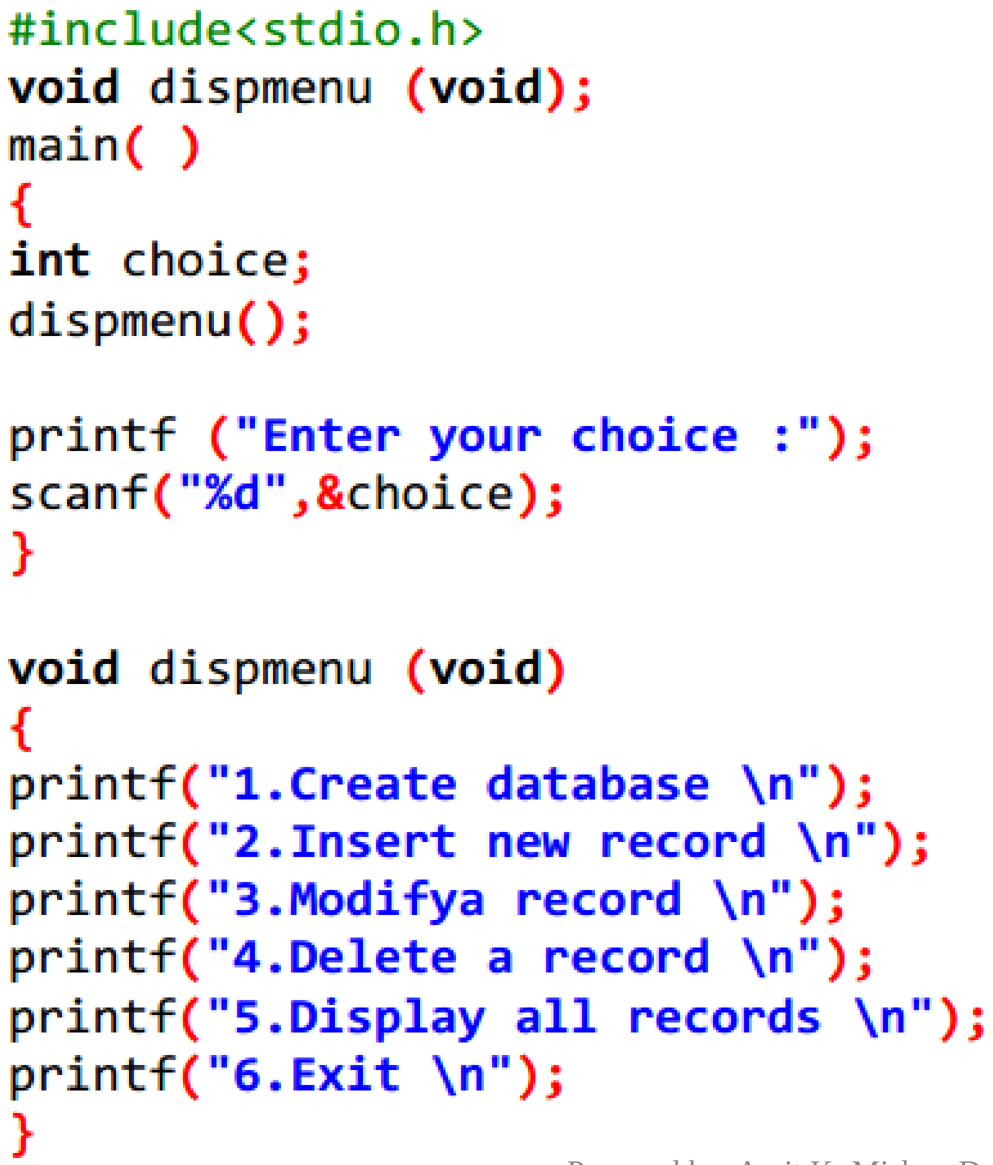
Functions that have no arguments and no return value can be written as:



```
void  func(void);
main()
{
        ........
        func();
        ........
}
void  func(void)
{
        ..........
        statement;
        ..........
}
```

- *In the this example, the function func( ) is called by main() and the function definition is written after the maine( ) function.*
- *As the function func( ) has no arguments, main( ) can not send any data to func( ) and since it has no return statement, hence function can not return any value to main().*
- *There is no communication between the calling and the called function.*

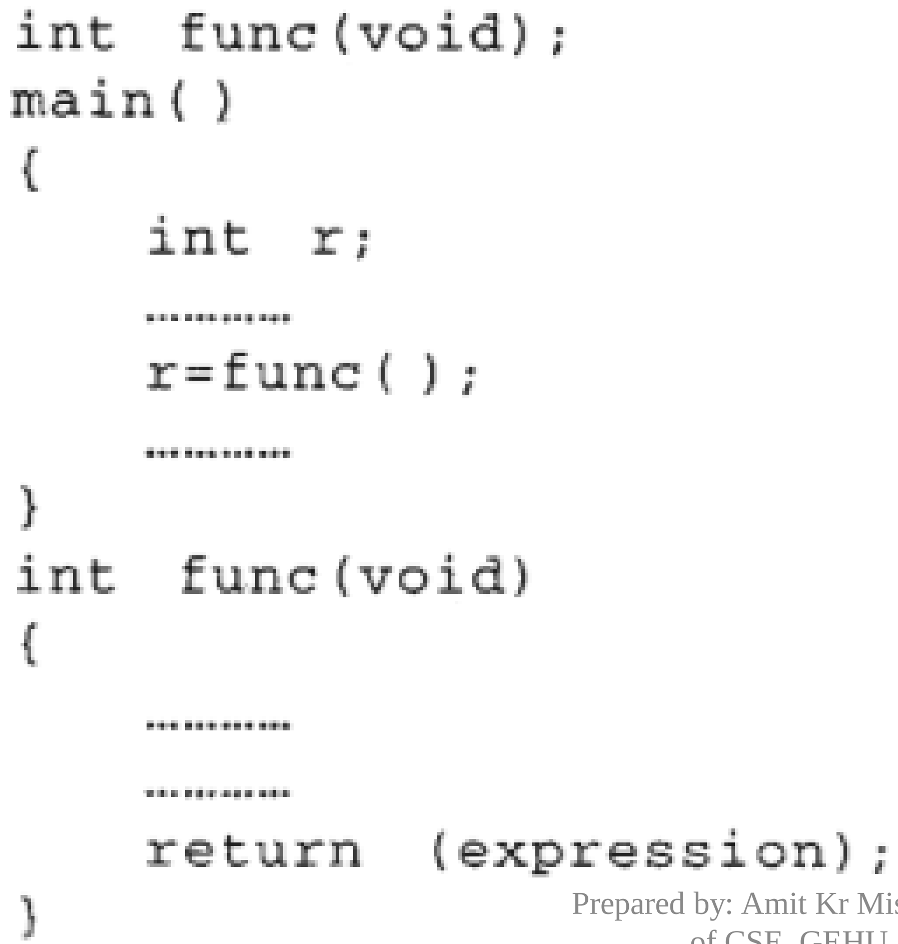
/*Program that uses a function with no arguments and no return



```
#include<stdio.h>
void dispmenu (void);
main( )
{
int choice;
dispmenu();

printf ("Enter your choice :");
scanf("%d",&choice);
}

void dispmenu (void)
{
printf("1.Create database \n");
printf("2.Insert new record \n");
printf("3.Modifya record \n");
printf("4.Delete a record \n");
printf("5.Display all records \n");
printf("6.Exit \n");
}
```

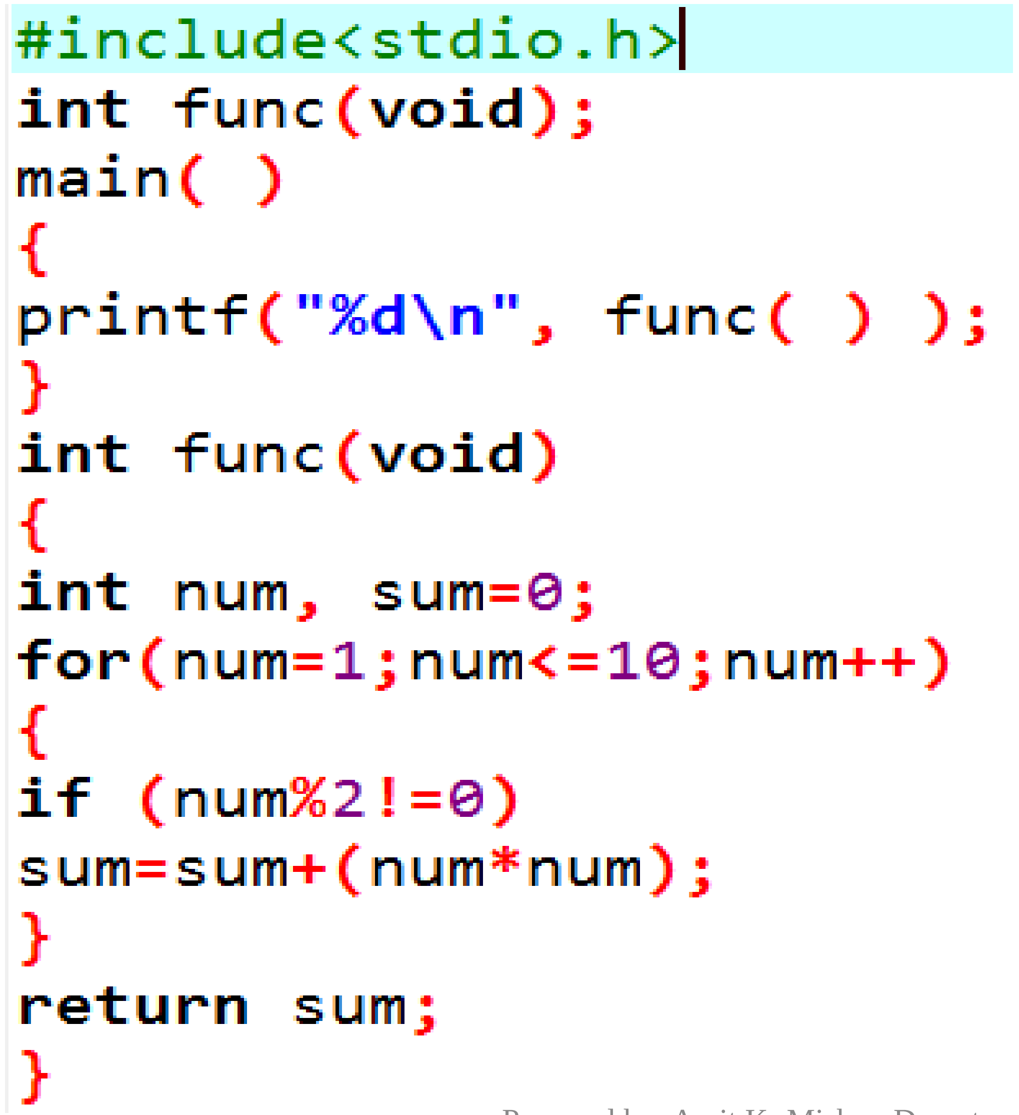
Function With No Arguments But A Return Value

These types of functions do not receive any arguments but they can return a value.



```
int  func(void);
main( )
{
     int  r;
     ...........
     r=func( );
     ...........
}
int  func(void)
{
     ...........
     ...........
     return  (expression);
}
```

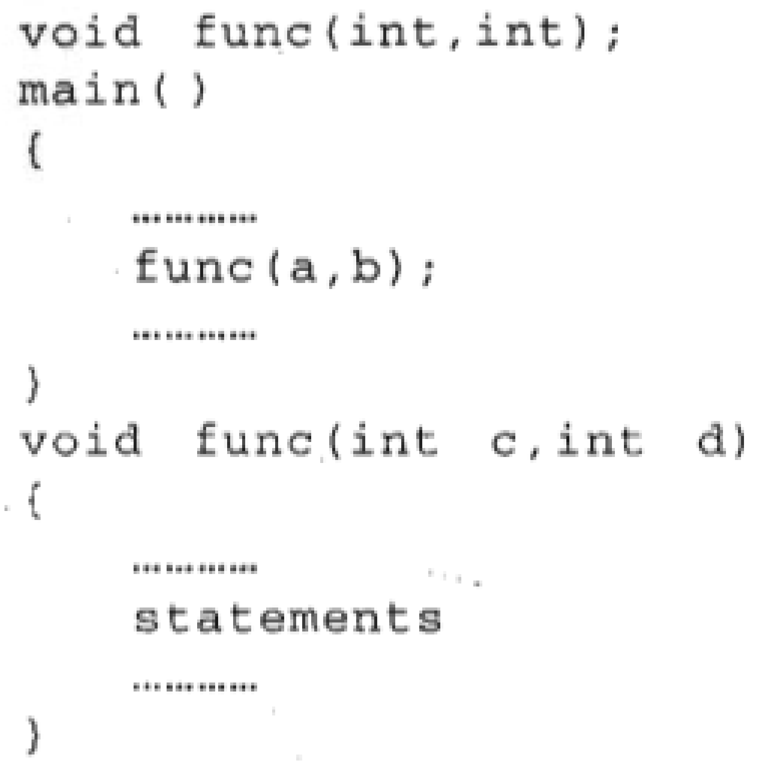
/*Program that returns the sum of squares of all odd numbers from 1 to 10*/



```c
#include<stdio.h>
int func(void);
main( )
{
printf("%d\n", func( ) );
}
int func(void)
{
int num, sum=0;
for(num=1;num<=10;num++)
{
if (num%2!=0)
sum=sum+(num*num);
}
return sum;
}
```

Function With Arguments But No Return Value

These types of functions have arguments, hence the calling function can send data to the called function but the called function does not return any value, These functions can be written as-



```
void  func(int,int);
main()
{
      ..........
      func(a,b);
      ..........
}
void  func(int  c,int  d)
{
      ..........
      statements
      ..........
}
```

Here a and b are actual arguments which are used for sending the values,

c and d are the formal arguments, which accept values from the actual arguments.

/*Program to find the type and area of a triangle.*/

```
#include<stdio.h>
#include<math.h>
void type(float a,float b, float c);
void area(float a,float b,float c);
main( )
{
float a,b,c;
printf ("Enter the sides of triangle: ");
scanf("%f%f%f",&a,&b,&c);
    if(a<b+c && b<c+a && c<a+b)
    {
    type (a,b,c);
    area(a,b,c);
    }
        else
        printf ("No triangle possible with these sides\n");
}
```

```
void type(float a, float b, float c)
    {
    if((a*a)+(b*b)==(c*c) ||(b*b)+(c*c)==(a*a) || (c*c)+(a*a)==(b*b))
    printf ("The triangle is right angled triangle\n");

    if(a==b && b==c)
    printf ("The triangle is equilateral \n");

          else if(a==b || b==c || c==a)
          printf("The triangle is isosceles\n");

    else
    printf ("The triangle is scalene\n");
    }

void area(float a,float b,float c)
    {
    float s, area;
    s= (a+b+c)/2;
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    printf ("The area of triangle = %f\n", area) ;
    }
```

Function With Arguments And Return Value

These types of functions have arguments, so the calling function can send data to the called function, it can also return any value to the calling function using return statement. This function can be written as :

```
int  func(int,int);
main( )
{
     int  r;
     …………
     r=func(a,b);
     …………
     func(c,d);
}
int  func(int  a,int  b )
{
     …………
     …………
     return  (expression);
}
```

/*Program to find the sum of digits of any number*/

```
#include<stdio.h>
int dsum(int n);
main()
{
int num,total;
printf("Enter the number ") ;
scanf("%d",&num);
total=dsum(num);
printf("Sum of digits is %d",total);
}

int dsum(int n)
{
int sum=0, rem;
    while(n>0)
    {
    rem=n%10;         /*rem takes the value of last digit*/
    sum=sum+rem;
    n=n/10;                     /* skips the last, digi t of number*/
    }
return (sum);
}
```

Recursion

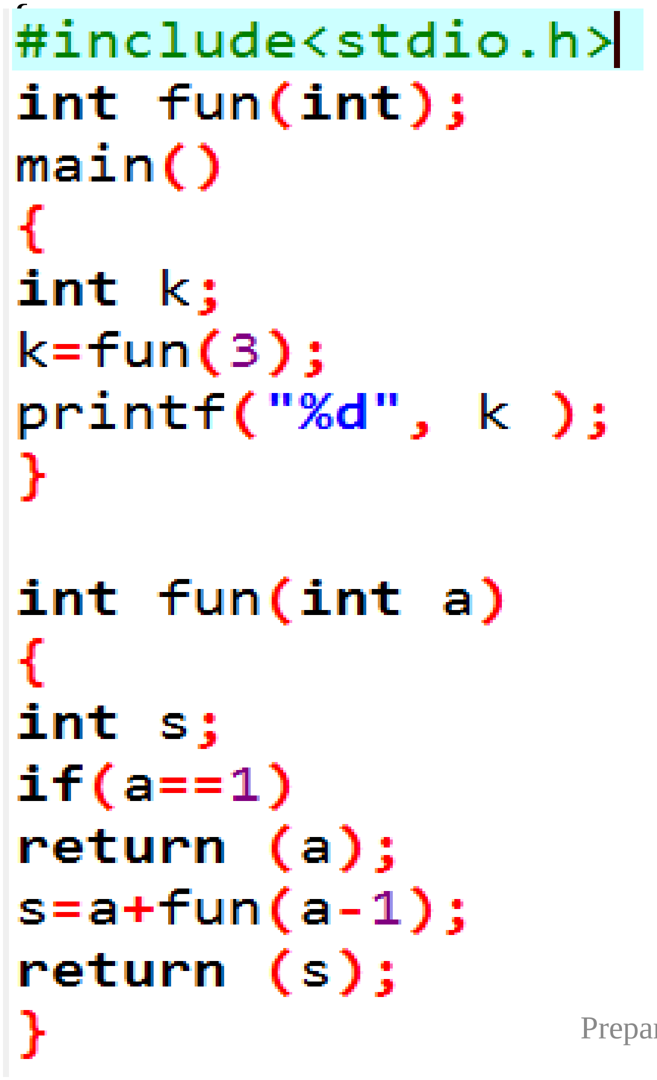
Function calling itself called recursion.

RECURSIVE FUNCTIONS

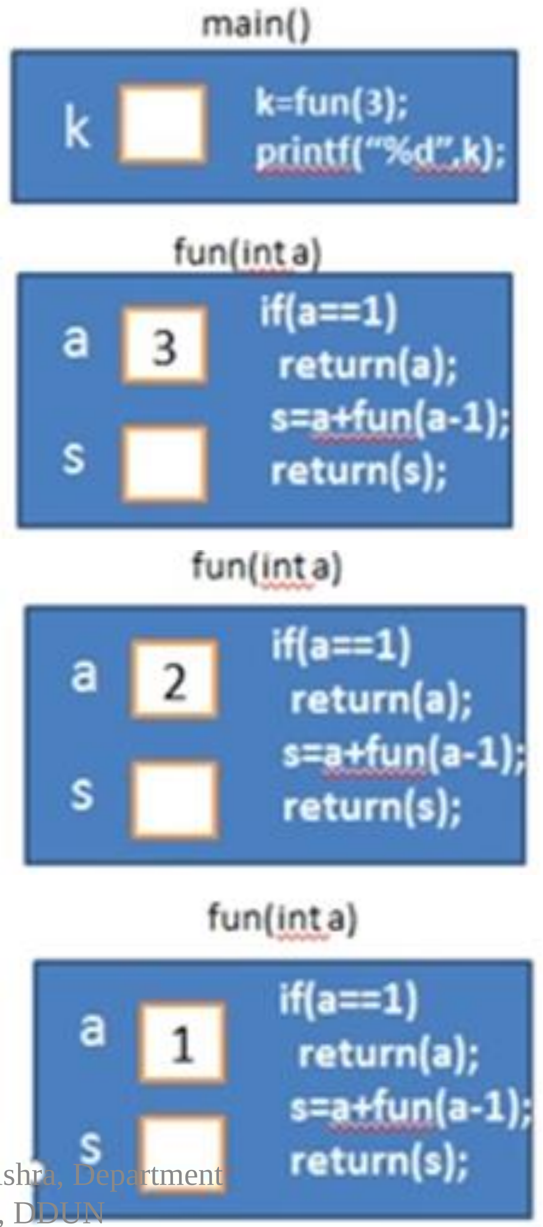
- A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- *Every recursive solution has two major cases, they are base case, in which the problem is simple enough to be solved directly without making any further calls to the same function*
  recursive case, in which first the problem at hand is divided into simpler sub parts. Second the function calls itself but with sub parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
- Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem. In recursive function, complicated problem is defined in terms of simpler problems and the simplest problem is given explicitly.

// Write a ‘C’ program to display sum of first 3 natural number using recursion.

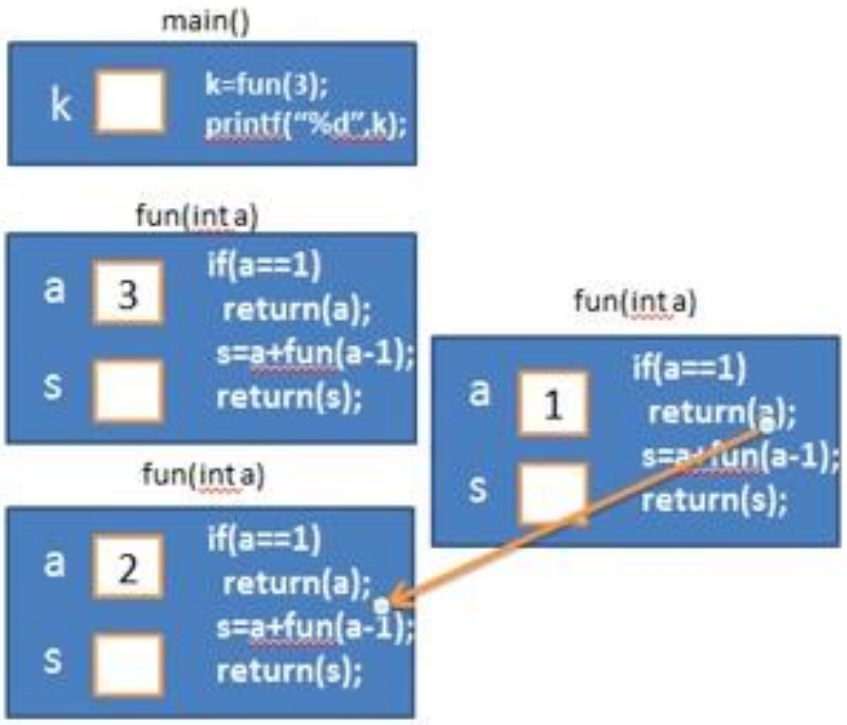
```
#include<stdio.h>
int fun(int);
main()
```



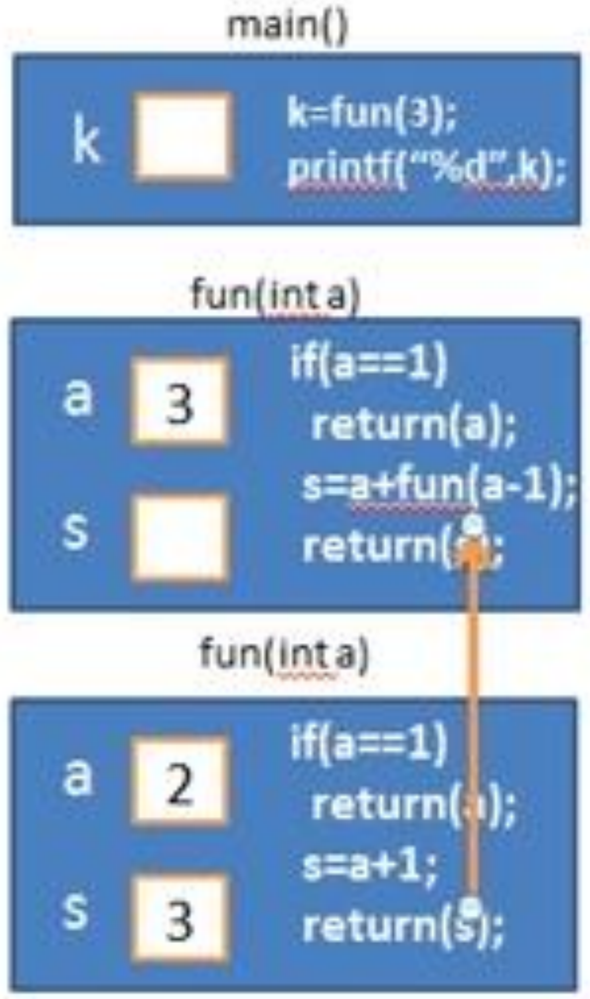
```
#include<stdio.h>
int fun(int);
main()
{
int k;
k=fun(3);
printf("%d", k );
}

int fun(int a)
{
int s;
if(a==1)
return (a);
s=a+fun(a-1);
return (s);
}
```



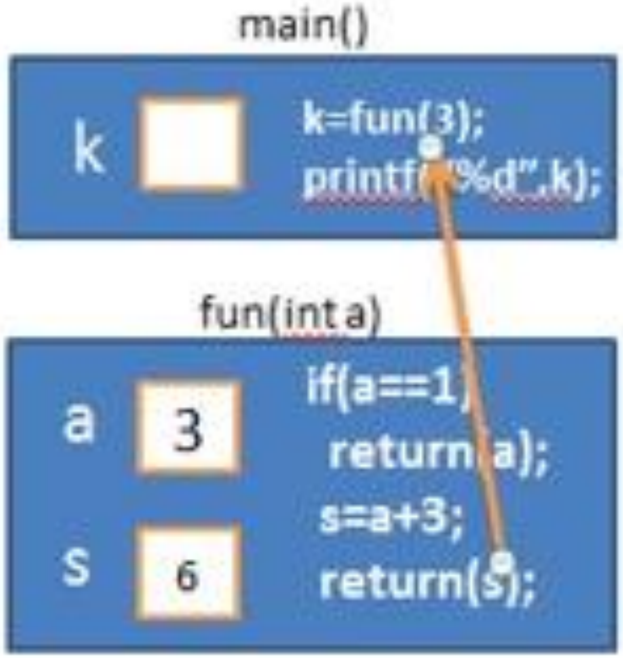
```
#include<stdio.h>
int fun(int);
main()
{
int k;
k=fun(3);
printf("%d", k );
}

int fun(int a)
{
int s;
if(a==1)
return (a);
s=a+fun(a-1);
return (s);
}
```



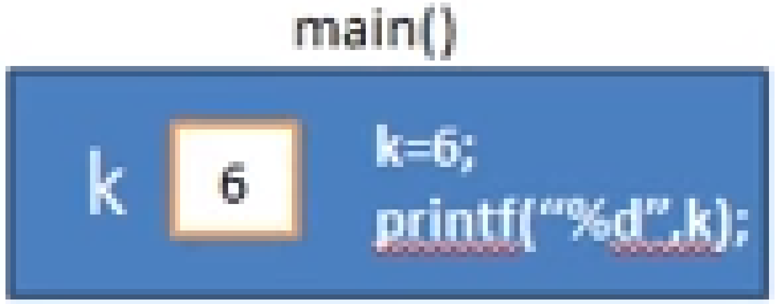
```c
#include<stdio.h>
int fun(int);
main()
{
int k;
k=fun(3);
printf("%d", k );
}

int fun(int a)
{
int s;
if(a==1)
return (a);
s=a+fun(a-1);
return (s);
}
```



```
#include<stdio.h>
int fun(int);
main()
{
int k;
k=fun(3);
printf("%d", k );
}

int fun(int a)
{
int s;
if(a==1)
return (a);
s=a+fun(a-1);
return (s);
}
```

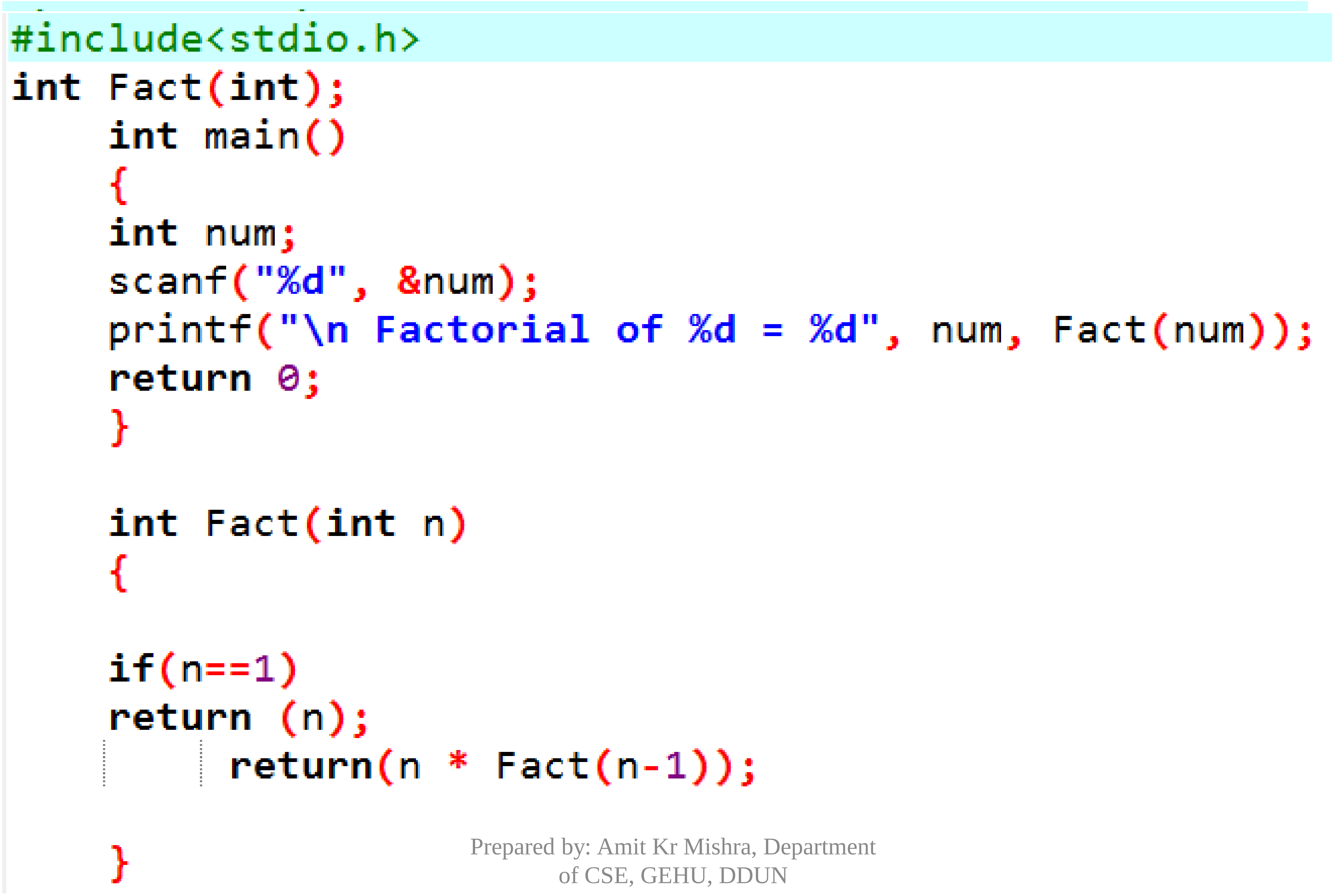


```c
#include<stdio.h>
int fun(int);
main()
{
int k;
k=fun(3);
printf("%d", k );
}

int fun(int a)
{
int s;
if(a==1)
return (a);
s=a+fun(a-1);
return (s);
}
```



#include<stdio.h>
int Fact(int);

// FINDING FACTORIAL OF A NUMBER USING RECURSION

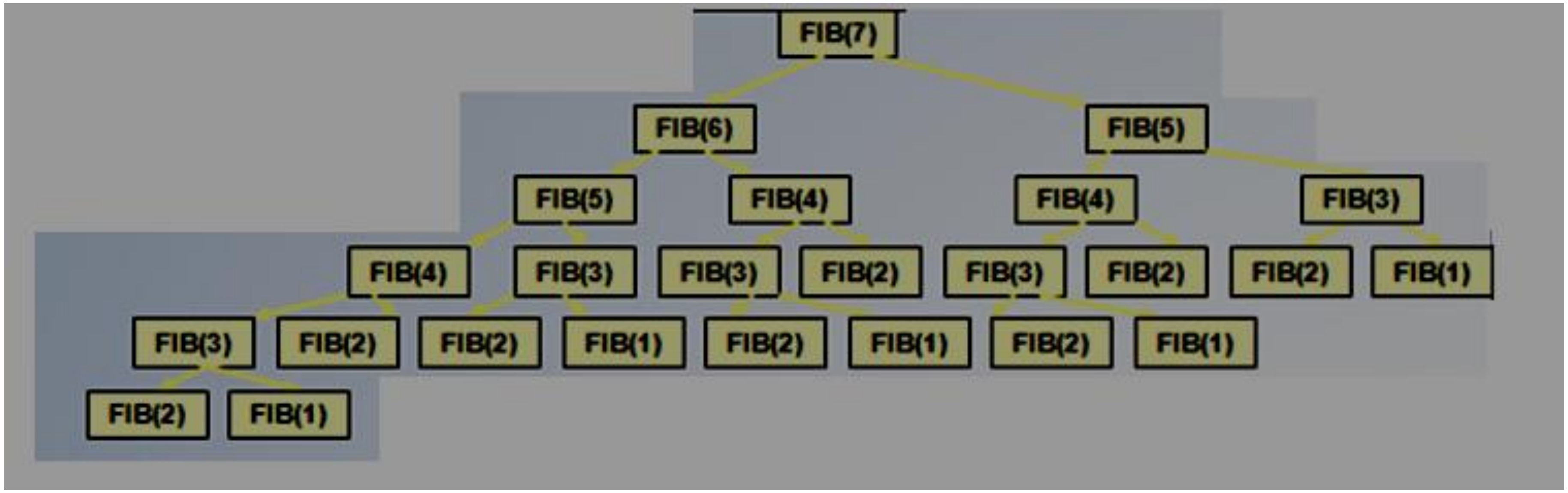


```
#include<stdio.h>
int Fact(int);
    int main()
    {
    int num;
    scanf("%d", &num);
    printf("\n Factorial of %d = %d", num, Fact(num));
    return 0;
    }

    int Fact(int n)
    {

    if(n==1)
    return (n);
        return(n * Fact(n-1));

    }
```

/* write a program for Fibonacci series. */

//THE FIBONACCI SERIES using recursion

```
#include<stdio.h>
int Fibonacci(int);
    void main()
    {
    int n,i;
    printf("\n Enter the number of terms in the series : ");
    scanf("%d", &n);
    for(i=1;i<=n;i++)
            printf(" %d",Fibonacci(i));
    }

    int Fibonacci(int num)
    {
            if(num==1 || num==2)
            return (1);
            return(Fibonacci(num-1)+Fibonacci(num-2));
    }
```


FIB(7)
FIB(6)
FIB(5)
FIB(5)
FIB(4)
FIB(4)
FIB(3)
FIB(4)
FIB(3)
FIB(3)
FIB(2)
FIB(3)
FIB(2)
FIB(2)
FIB(1)
FIB(3)
FIB(2)
FIB(2)
FIB(1)
FIB(2)
FIB(1)
FIB(2)
FIB(1)
FIB(2)
FIB(1)

Call by value

```
#include<stdio.h>
void swap(int a, int b);
     void main()
     {
     int a, b;
   printf("\n Enter the numbers to be swapped: ");
     scanf("%d %d", &a, &b);
     printf("\n The numbers before swapping a=%d and b=%d", a, b);
     swap(a,b);
     }


     void swap(int a, int b)
     {
     a = a+b;
     b = a-b;
     a = a-b;
     printf("\n\n The numbers after swapping a=%d and b=%d",a,b);
     }
```

Call by reference

```
#include<stdio.h>
void swap(int *a, int *b);
      void main()
      {
      int x, y;
    printf("\n Enter the numbers to be swapped: ");
      scanf("%d %d", &x, &y);

      swap(&x,&y);
      }


      void swap(int *a, int *b)
      {
      int temp;
      temp = *a;
      a = *b;
      b = temp;
      printf("\n\n The numbers after swapping a=%d and b=%d",a,b);
      }
```

//Write a program to pass array in function

```
#include<stdio.h>
void display(int a[])
{
int i;
for(i=0;i<5;i++)
{
printf("%d ",a[i]);
}

}
void main()
{
int a[5];
int i;
printf("enter five elements\n");
for(i=0;i<5;i++)
{
scanf("%d",&a[i]);
}
display(a);

}
```

Storage classes in C Language

$\underline{\text{in}}\text{t}\ \underline{\text{x}}\text{=5;}$

$$\overset{x}{\boxed{\;5\;}}$$

① x

② $\xrightarrow{\qquad\qquad}$ 4 bytes

③ Datatype

Characteristics of a variable

```
int x=5;
```

- Three properties of variables
  - Name of variable
  - Size of memory block
  - Type of content

Other properties of variable

- Default Value
- Storage
- Scope
- Life

Storage classes

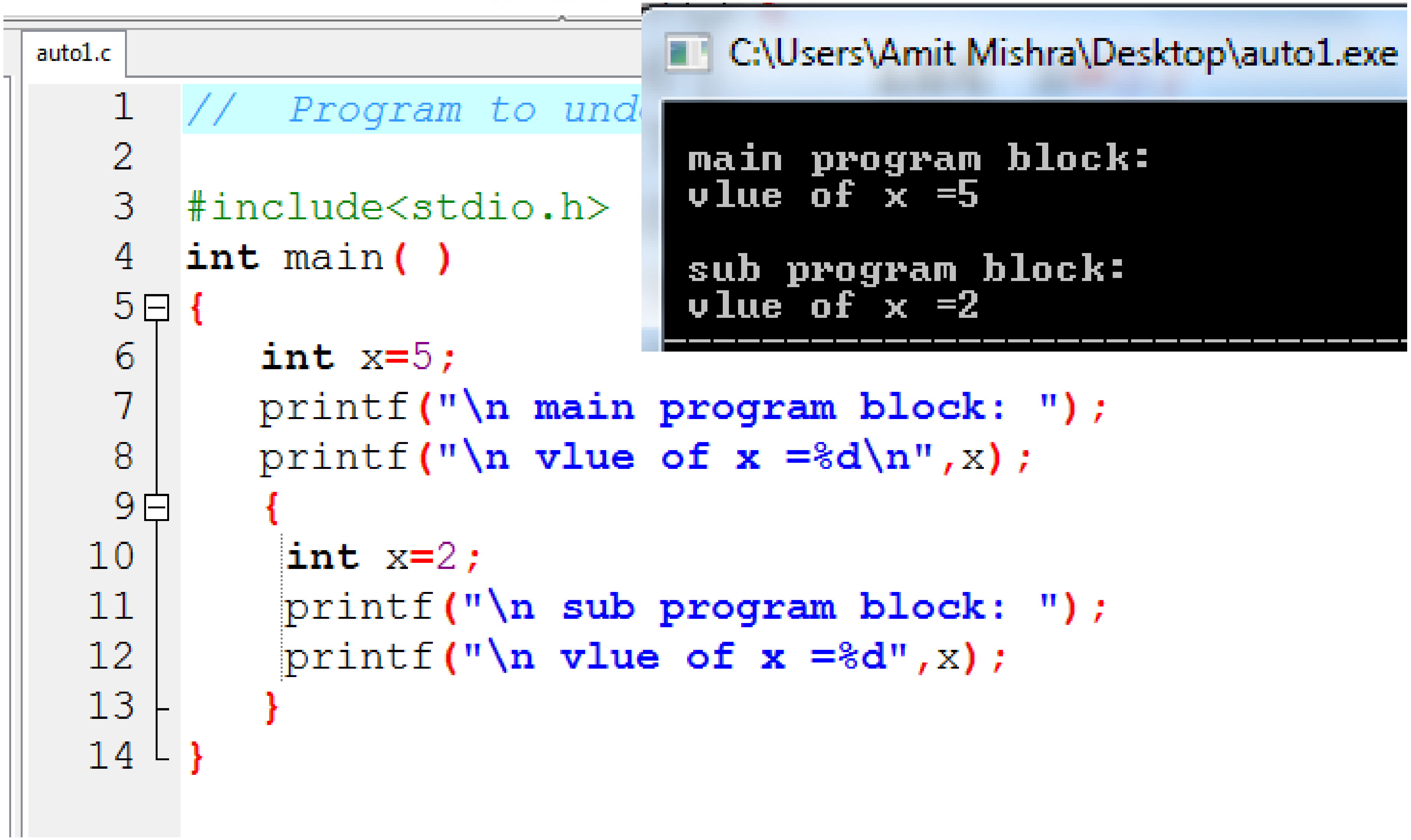
- Automatic
- Register
- Static
- External

Storage Class	Keyword	Default Value	Storage	Scope	Life
Automatic	**auto**	Garbage	RAM	Limited to the block in which it is declared	Till the execution of the block in which it is declared
Register	**register**	Garbage	register	Same	same
Static	**static**	0 (zero)	RAM	Same	Till the end of program
External	**extern**	0 (zero)	RAM	Global	Same

Automatic:

- All the variables declared inside a block without any storage class specifier are called automatic variables.
- The keyword **auto** used to declare automatic variables.
- The uninitialized automatic variables initially contain garbage value.
- The scope of these variables is inside the function or block in which they are declared and they can't be used in any other function/block.
- They are named automatic since storage for them is reserved automatically each time when the control enters the function/block and are released automatically when the function/block terminates.

- Automatic



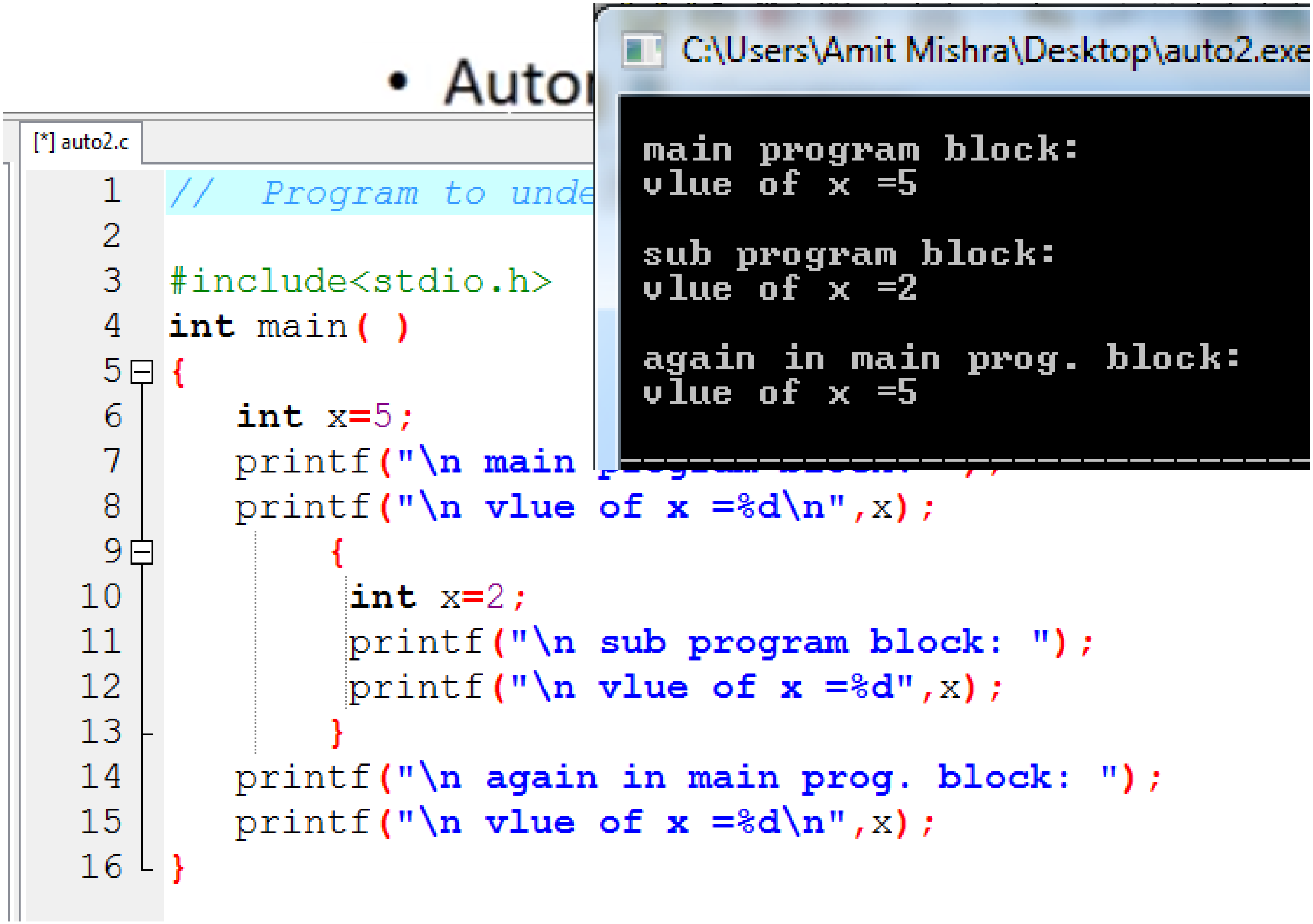
auto1.c

```
//   Program to und

#include<stdio.h>
int main( )
{
    int x=5;
    printf("\n main program block: ");
    printf("\n vlue of x =%d\n",x);
    {
     int x=2;
     printf("\n sub program block: ");
     printf("\n vlue of x =%d",x);
    }
}
```

C:\Users\Amit Mishra\Desktop\auto1.exe

```
main program block:
vlue of x =5

sub program block:
vlue of x =2
```

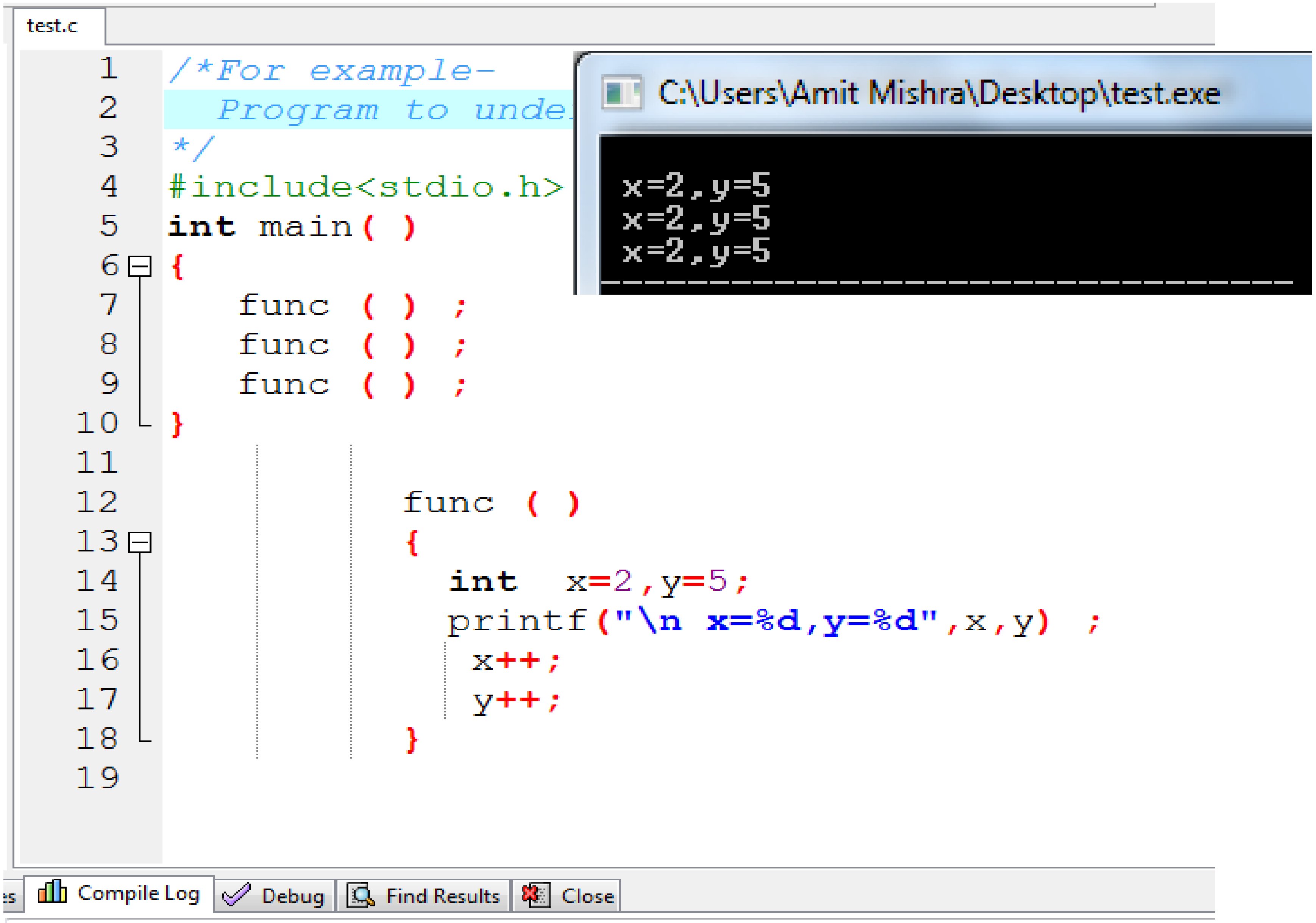
• Autor



```c
//   Program to unde

#include<stdio.h>
int main( )
{
    int x=5;
    printf("\n main
    printf("\n vlue of x =%d\n",x);
        {
         int x=2;
         printf("\n sub program block: ");
         printf("\n vlue of x =%d",x);
        }
    printf("\n again in main prog. block: ");
    printf("\n vlue of x =%d\n",x);
}
```

```
main program block:
vlue of x =5

sub program block:
vlue of x =2

again in main prog. block:
vlue of x =5
```

- Automatic



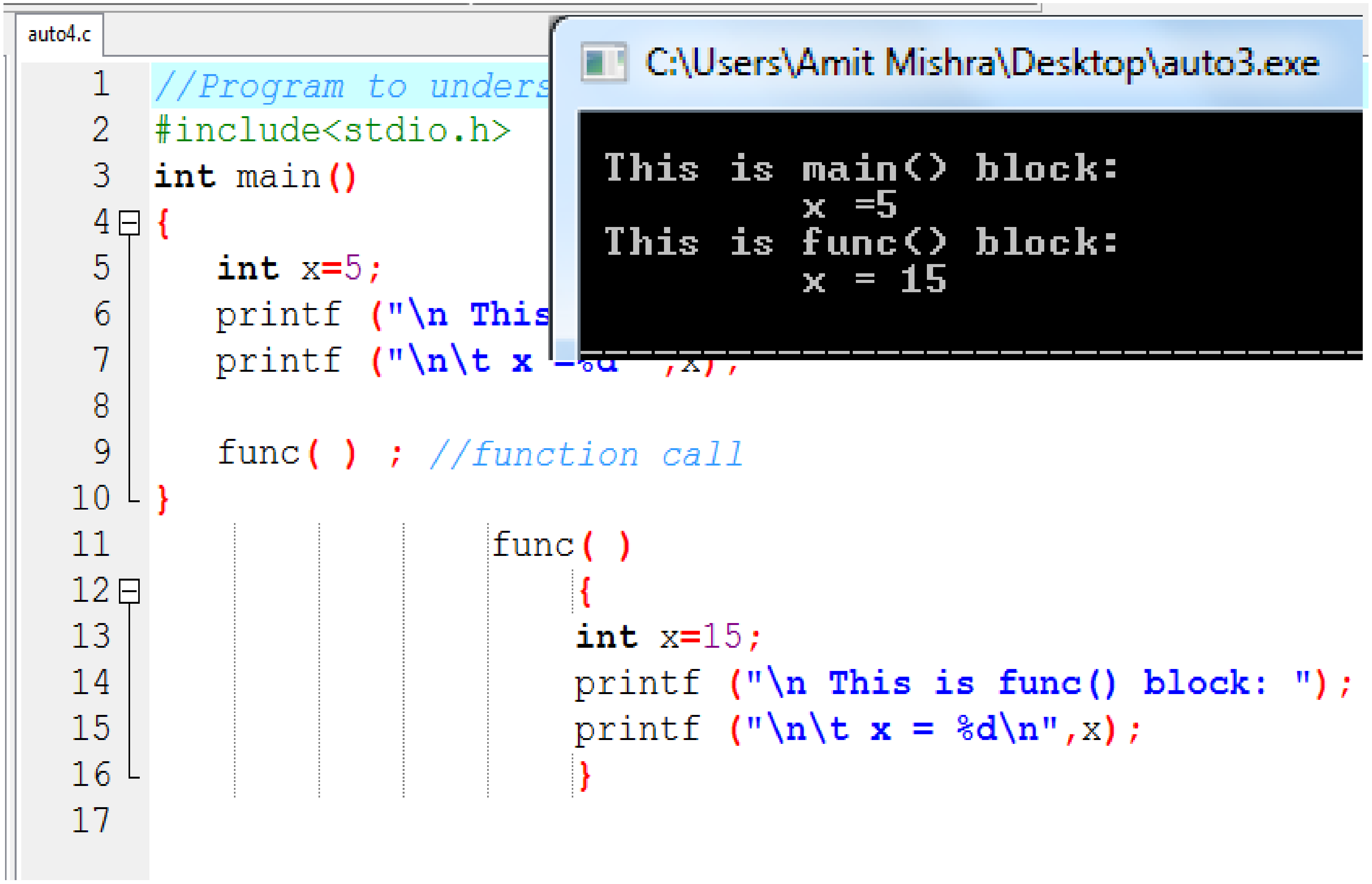
test.c

```
/*For example-
  Program to unde
*/
#include<stdio.h>
int main( )
{
    func ( ) ;
    func ( ) ;
    func ( ) ;
}

              func ( )
              {
                int  x=2,y=5;
                printf("\n x=%d,y=%d",x,y) ;
                 x++;
                 y++;
              }
```

C:\Users\Amit Mishra\Desktop\test.exe

```

x=2,y=5
x=2,y=5
x=2,y=5
```

- Automatic



auto4.c

```
//Program to unders
#include<stdio.h>
int main()
{
    int x=5;
    printf ("\n This
    printf ("\n\t x =%d  ",x);

    func( ) ; //function call
}
                    func( )
                    {
                    int x=15;
                    printf ("\n This is func() block: ");
                    printf ("\n\t x = %d\n",x);
                    }
```

C:\Users\Amit Mishra\Desktop\auto3.exe

```
This is main() block:
        x =5
This is func() block:
        x = 15
```

External:

- The variables that have to be used by many functions and different files can be declared as as external variable.
- The initial value of an uninitialized external variable is zero.
- The keyword **extern** is specified in declaration but not in definition.

For example:

- The definition of an external variable salary can be written as-

  float salary;

- Its declaration will be written as-

  extern float salary;

External:

Definition of an external variable-

for example: **float salary;**

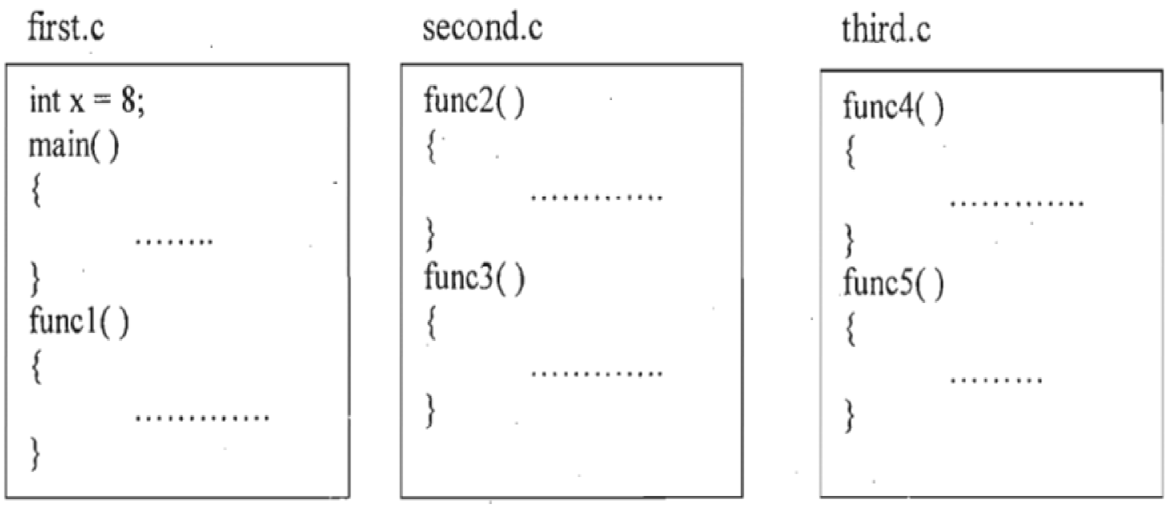
- Definition creates the variable, so memory is allocated at the time of definition.
- There can be only one definition.
- The keyword extern is not specified in the definition

Declaration of an external variable-

for example: **extern float salary;**

- The declaration does not create the variable, it only refers to a variable that has already been created somewhere, so memory is not allocated at the time of declaration.
- There can be many declarations.
- The variable cannot be initialized at the time of declaration.

- When the program is large it is written in different files and these files are compiled separately and linked together afterwards to form an executable program.
- Now consider a multifile program which is written in three files viz. first.c, second.c and third.c.



first.c

```
int x = 8;
main( )
{
        ........
}
func1( )
{
        .............
}
```

second.c

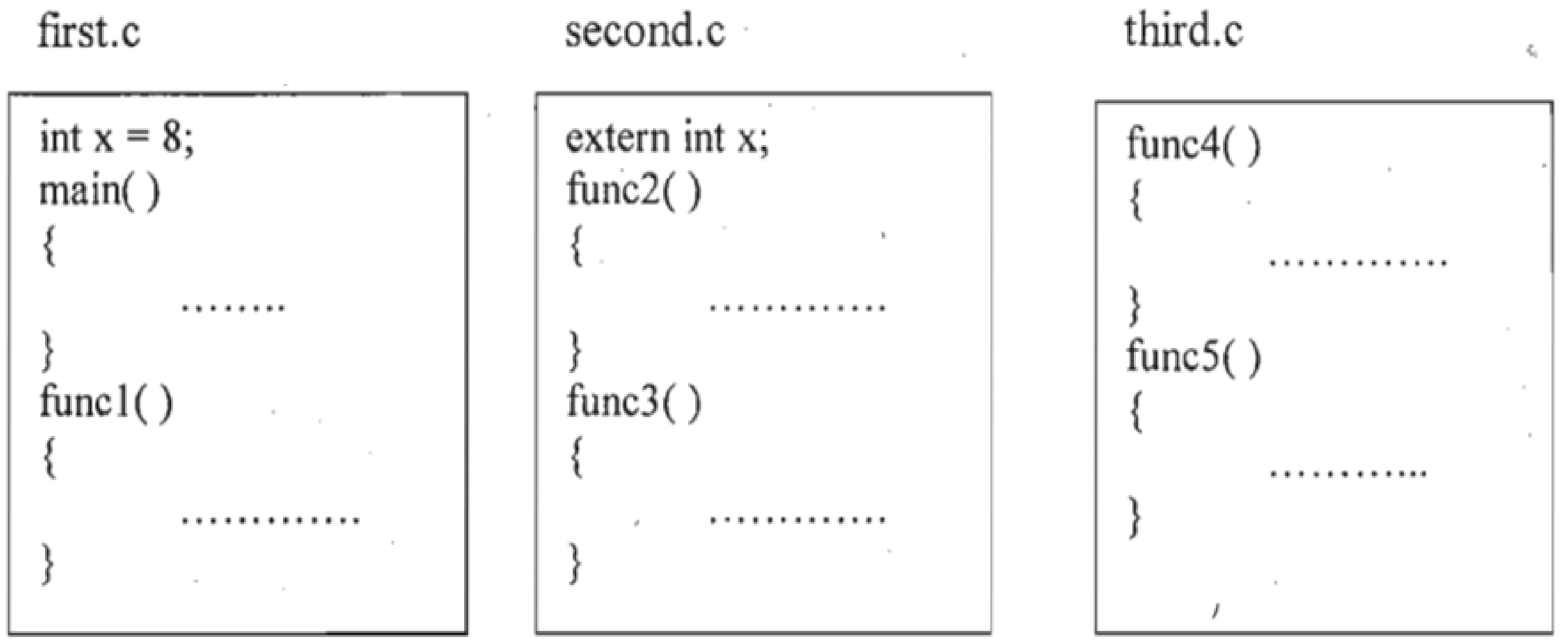
```
func2( )
{
        .............
}
func3( )
{
        .............
}
```

third.c

```
func4( )
{
        .............
}
func5( )
{
        .........
}
```

- Here in the file first.c, an external variable x is defined and initialized. This variable can be used both in main( ) and func1( ) but it is not accessible to other files i.e. second.c and third.c.

Suppose file second.c wants to access this variable then we can put the declaration in this file as:


first.c

```
int x = 8;
main( )
{
        ........
}
func1( )
{
        .............
}
```

second.c

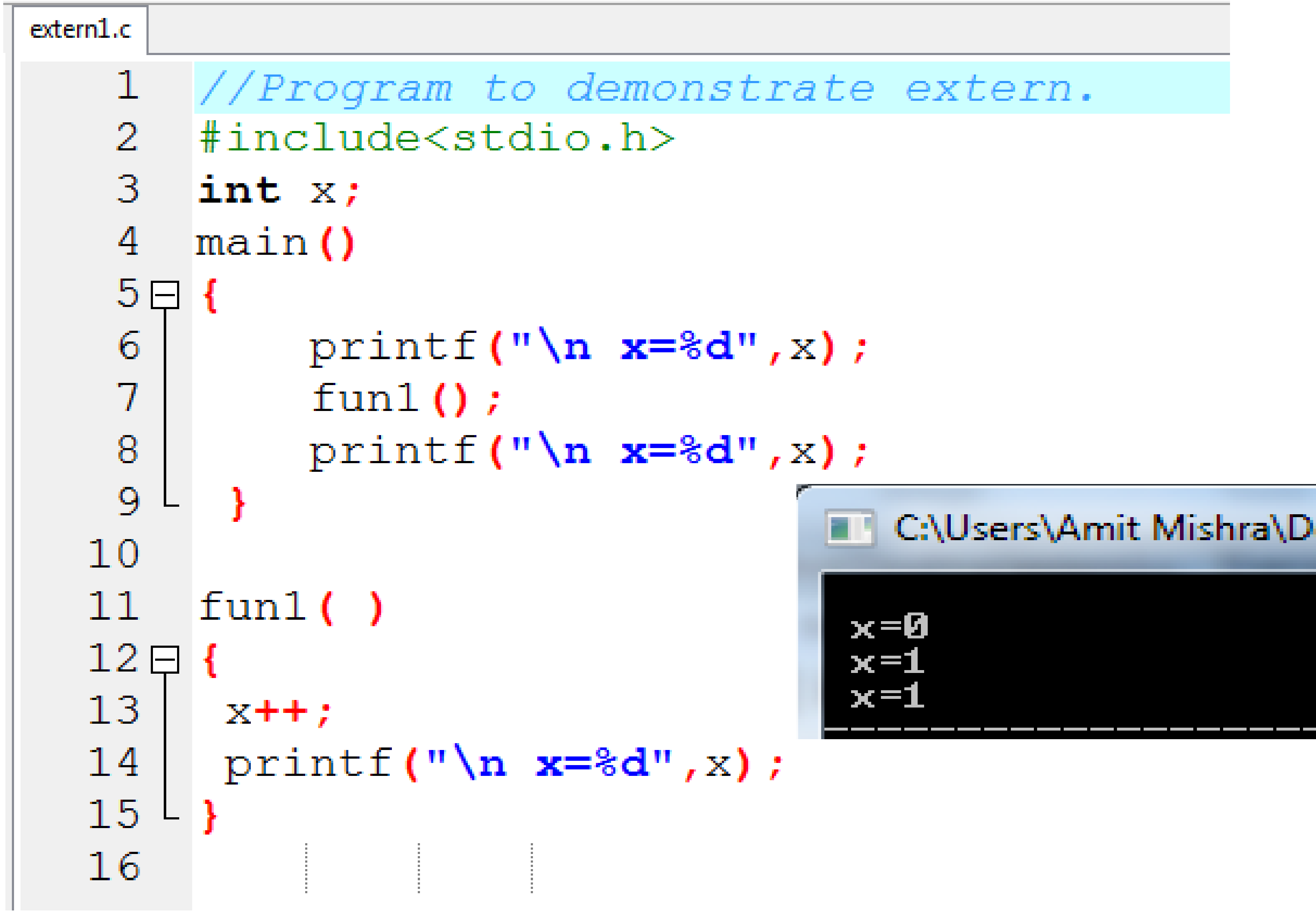
```
extern int x;
func2( )
{
        .............
}
func3( )
{
        .............
}
```

third.c

```
func4( )
{
        .............
}
func5( )
{
        ...........
}
```

irst.c
both

- External

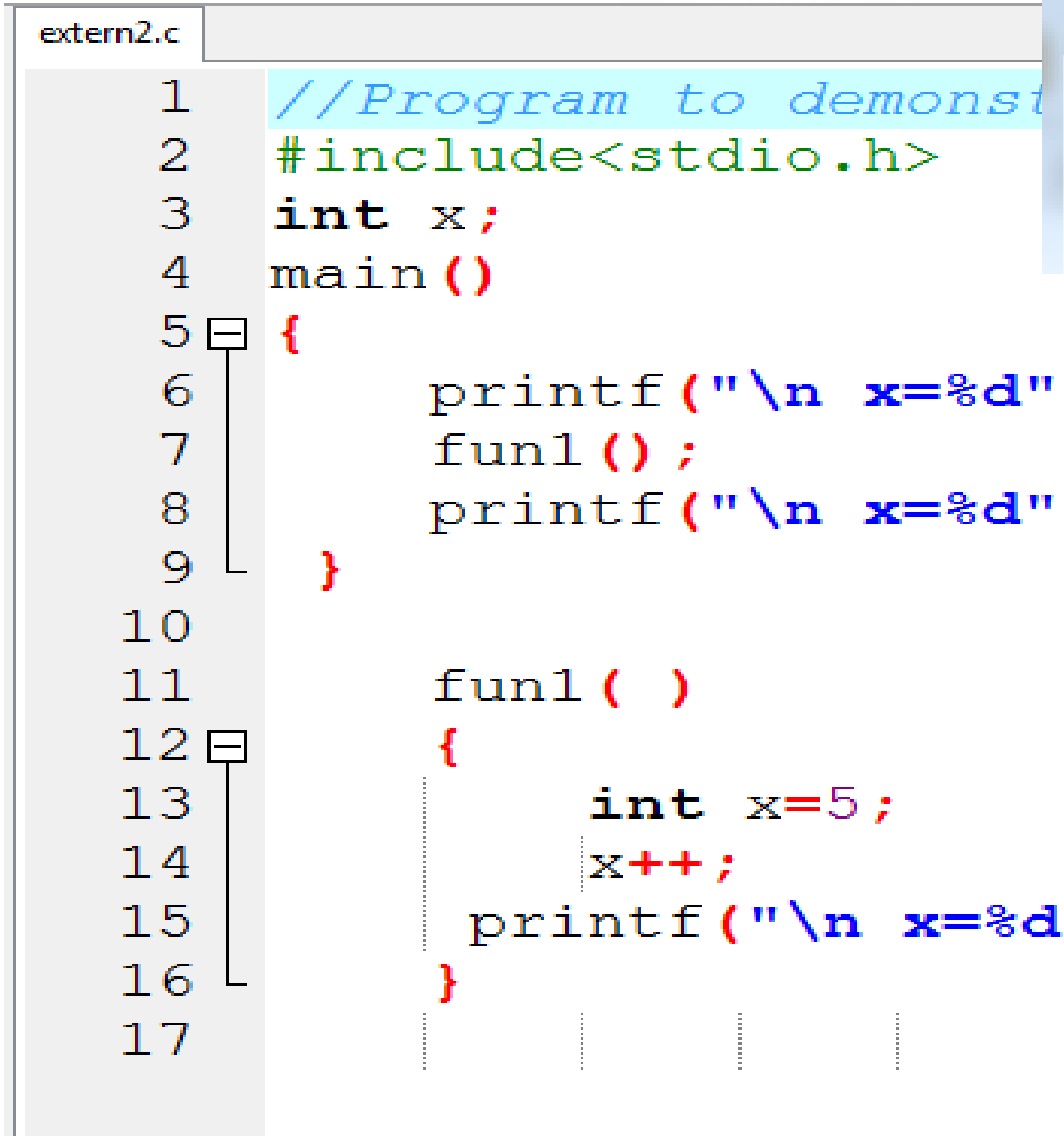

extern1.c

```c
//Program to demonstrate extern.
#include<stdio.h>
int x;
main()
{
    printf("\n x=%d",x);
    fun1();
    printf("\n x=%d",x);
 }

fun1( )
{
 x++;
 printf("\n x=%d",x);
}

```

C:\Users\Amit Mishra\D

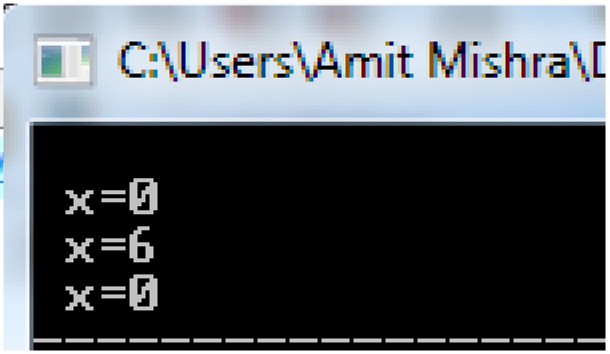
```
 x=0
 x=1
 x=1
```

- External



extern2.c

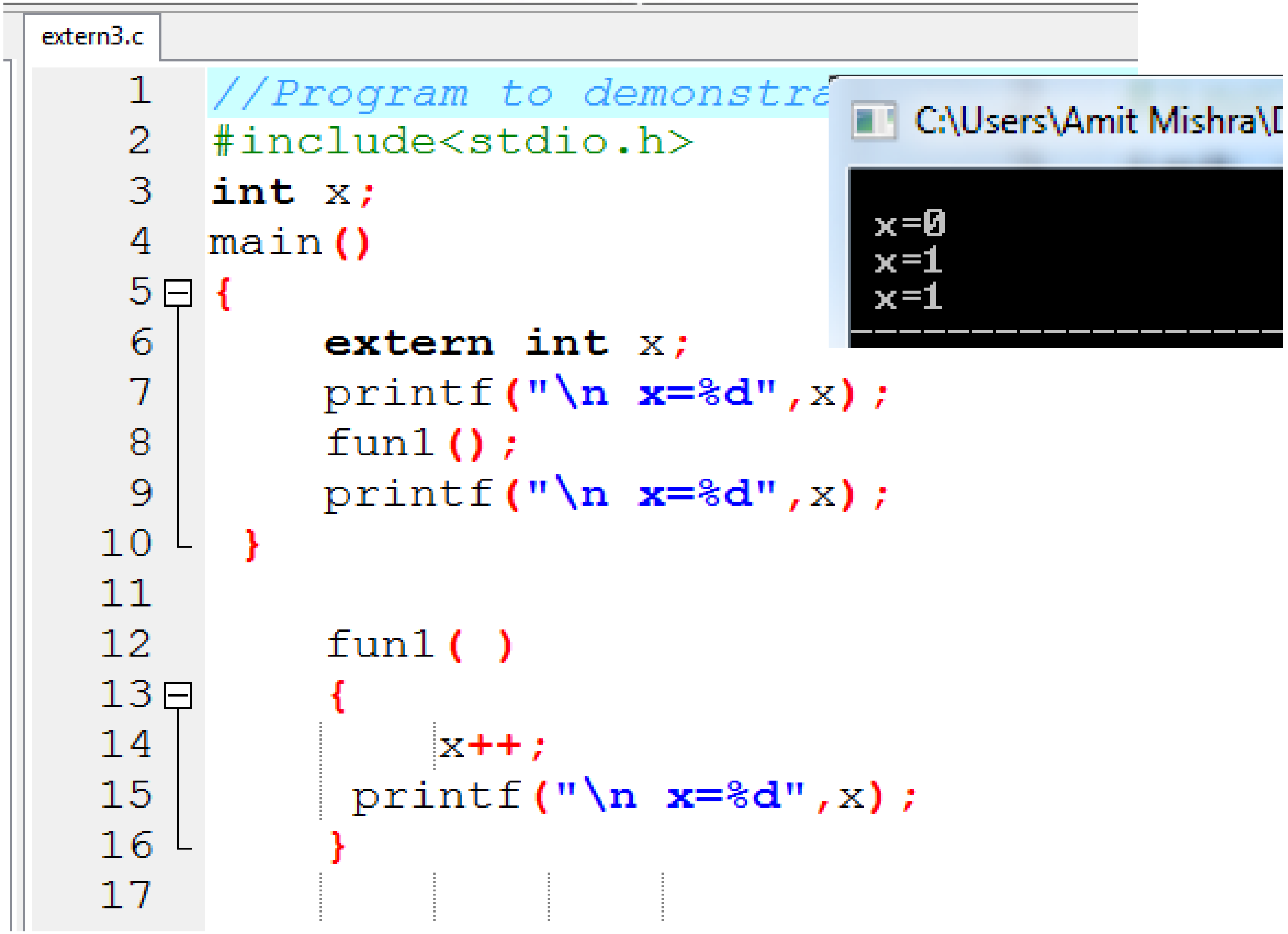
```c
//Program to demonst
#include<stdio.h>
int x;
main()
{
        printf("\n x=%d",x);
        fun1();
        printf("\n x=%d",x);
  }

        fun1( )
        {
                int x=5;
                x++;
          printf("\n x=%d",x);
        }
```



C:\Users\Amit Mishra\D

```
 x=0
 x=6
 x=0
```

- External



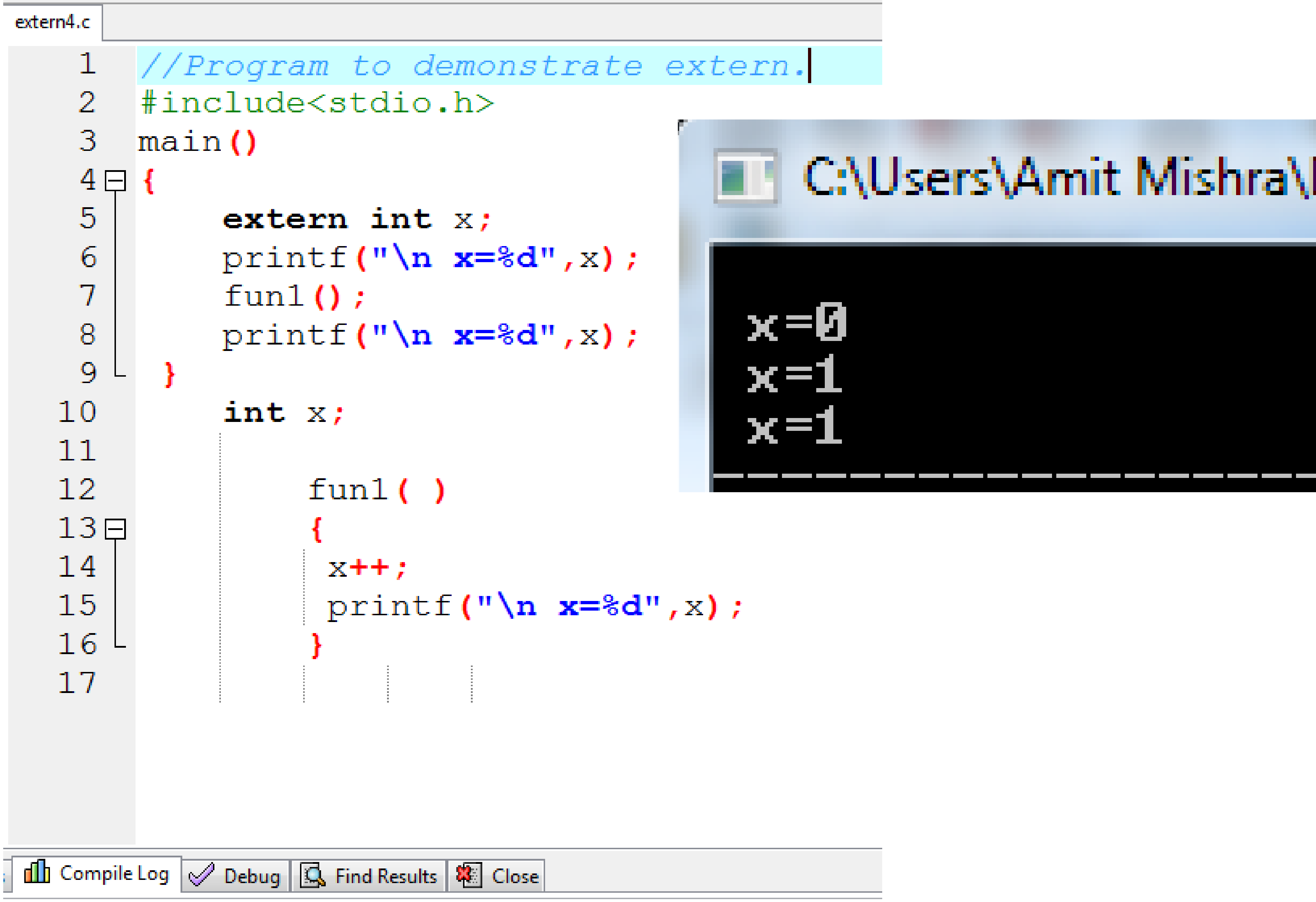
extern3.c

```
//Program to demonstra
#include<stdio.h>
int x;
main()
{
	extern int x;
	printf("\n x=%d",x);
	fun1();
	printf("\n x=%d",x);
 }

	fun1( )
	{
		x++;
	 printf("\n x=%d",x);
	}

```

C:\Users\Amit Mishra\D

```
x=0
x=1
x=1
```

- External



extern4.c

```
//Program to demonstrate extern.
#include<stdio.h>
main()
{
     extern int x;
     printf("\n x=%d",x);
     fun1();
     printf("\n x=%d",x);
 }
     int x;

          fun1( )
          {
           x++;
           printf("\n x=%d",x);
          }

```

C:\Users\Amit Mishra\

```
 x=0
 x=1
 x=1
```

Static

There are two types of static variables-

(1) Local static variables

- (2) Global static variables

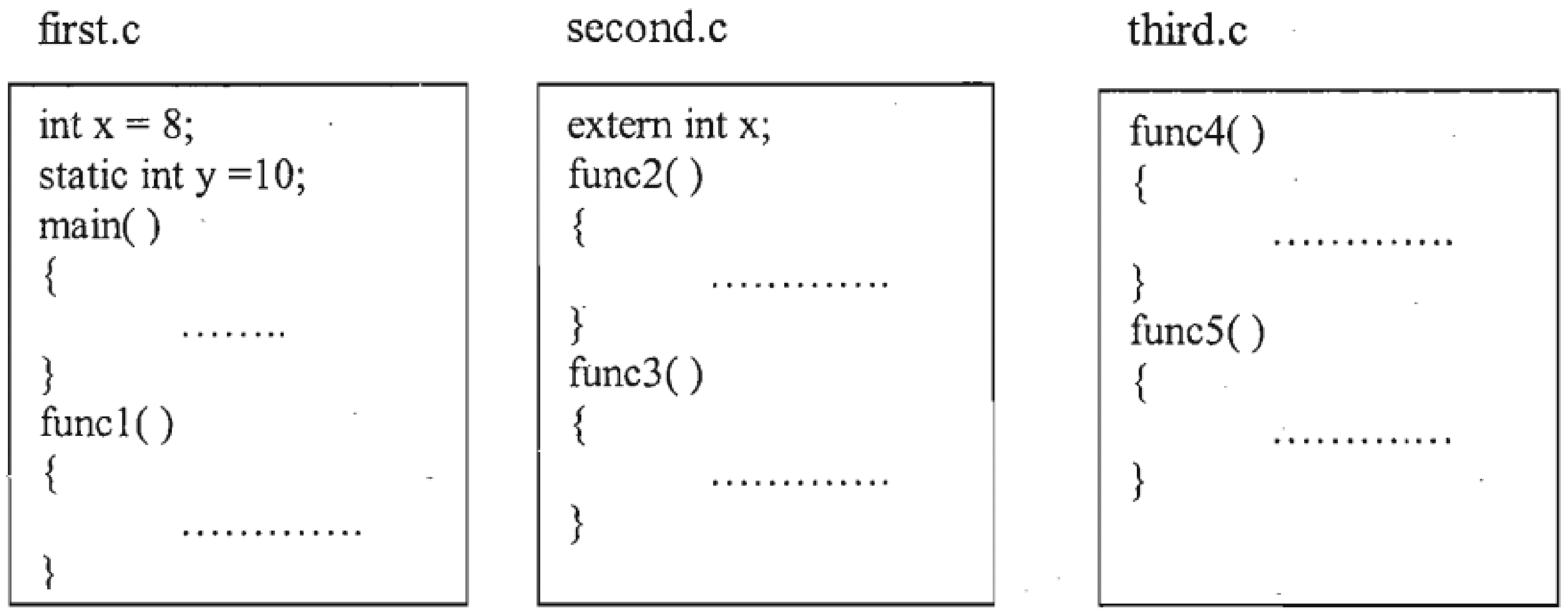
- **Local Static Variables:**
  The scope of a local static variable is same as that of an automatic variable.
- A static variable is created at the compilation time and it remains alive till the end of a program.
- A static variable is created only once.
- If it has been initialized, then the initialization value is placed in it only once at the time of creation.
- It is not initialized each time the function is called.
- A static variable can be initialized only by constants. If a static variable is not explicitly initialized then by default it takes initial value zero.

Global Static Variables

In the case of global variables the static specifier is not used to extend the lifetime since global variables have already a lifetime equal to the life of program.

If an external variable is defined as static, then it can't be used by other files of the program. So we can make an external variable private to a file by making it static.

Here the variable y is defined as a static external variable, so it can be used only in the file first.c. We can't use it in other files by putting extern declaration for it.



first.c

```
int x = 8;
static int y =10;
main( )
{
        ........
}
func1( )
{
        .............
}
```

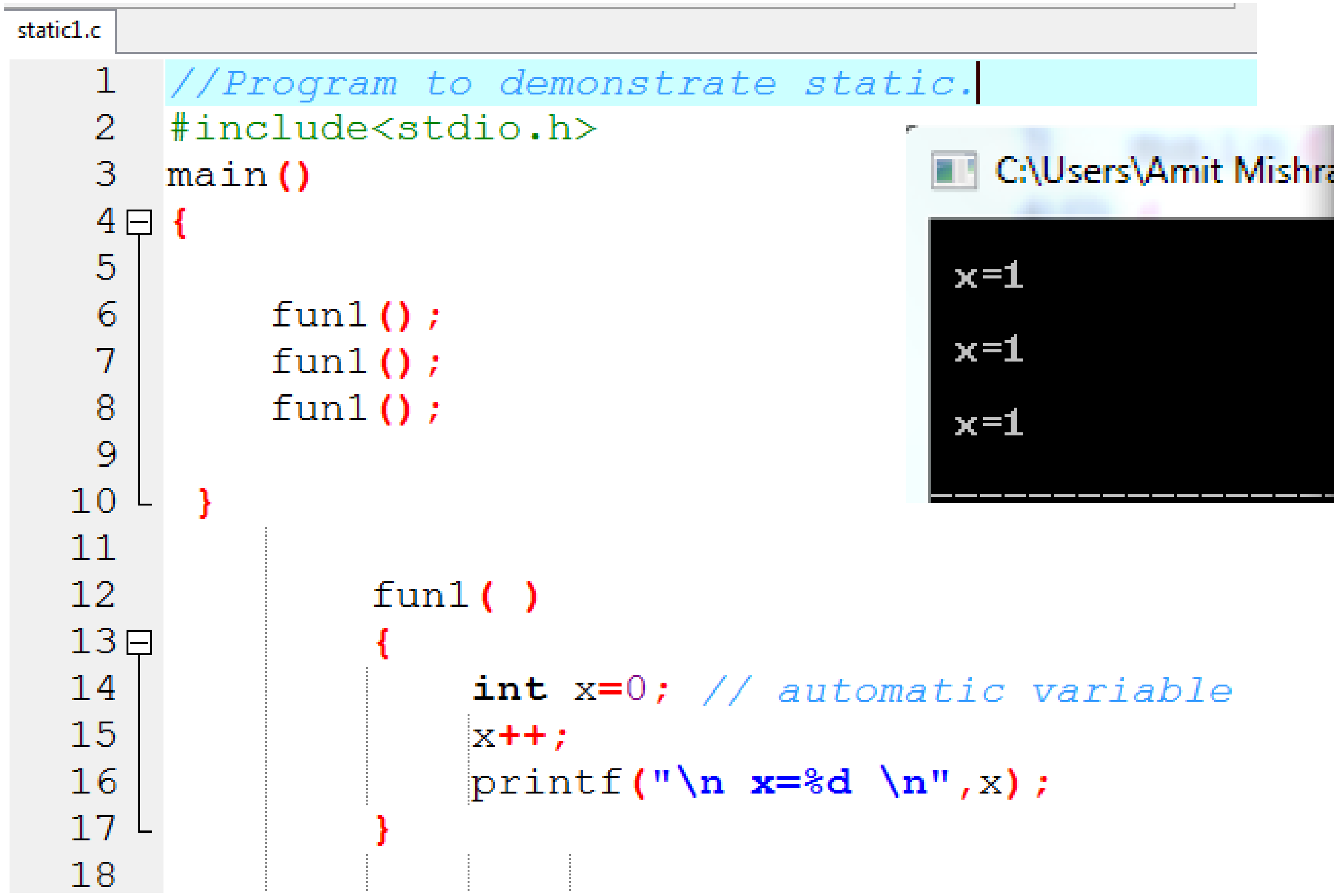
second.c

```
extern int x;
func2( )
{
        .............
}
func3( )
{
        .............
}
```

third.c

```
func4( )
{
        .............
}
func5( )
{
        .............
}
```

- Static



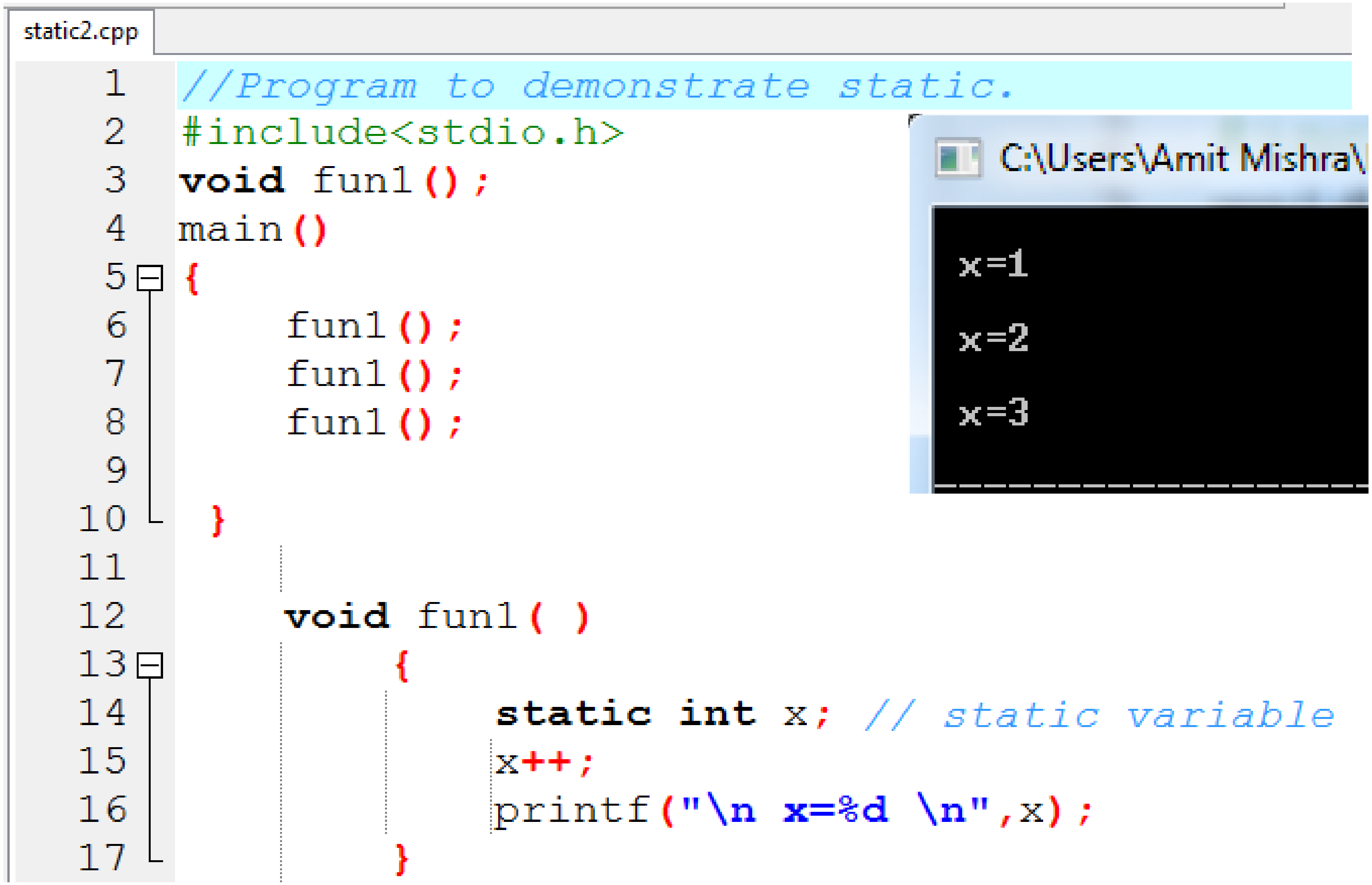
static1.c

```
//Program to demonstrate static.
#include<stdio.h>
main()
{

     fun1();
     fun1();
     fun1();

 }

          fun1( )
          {
               int x=0; // automatic variable
               x++;
               printf("\n x=%d \n",x);
          }

```

C:\Users\Amit Mishra

```
x=1

x=1

x=1
```

- Static



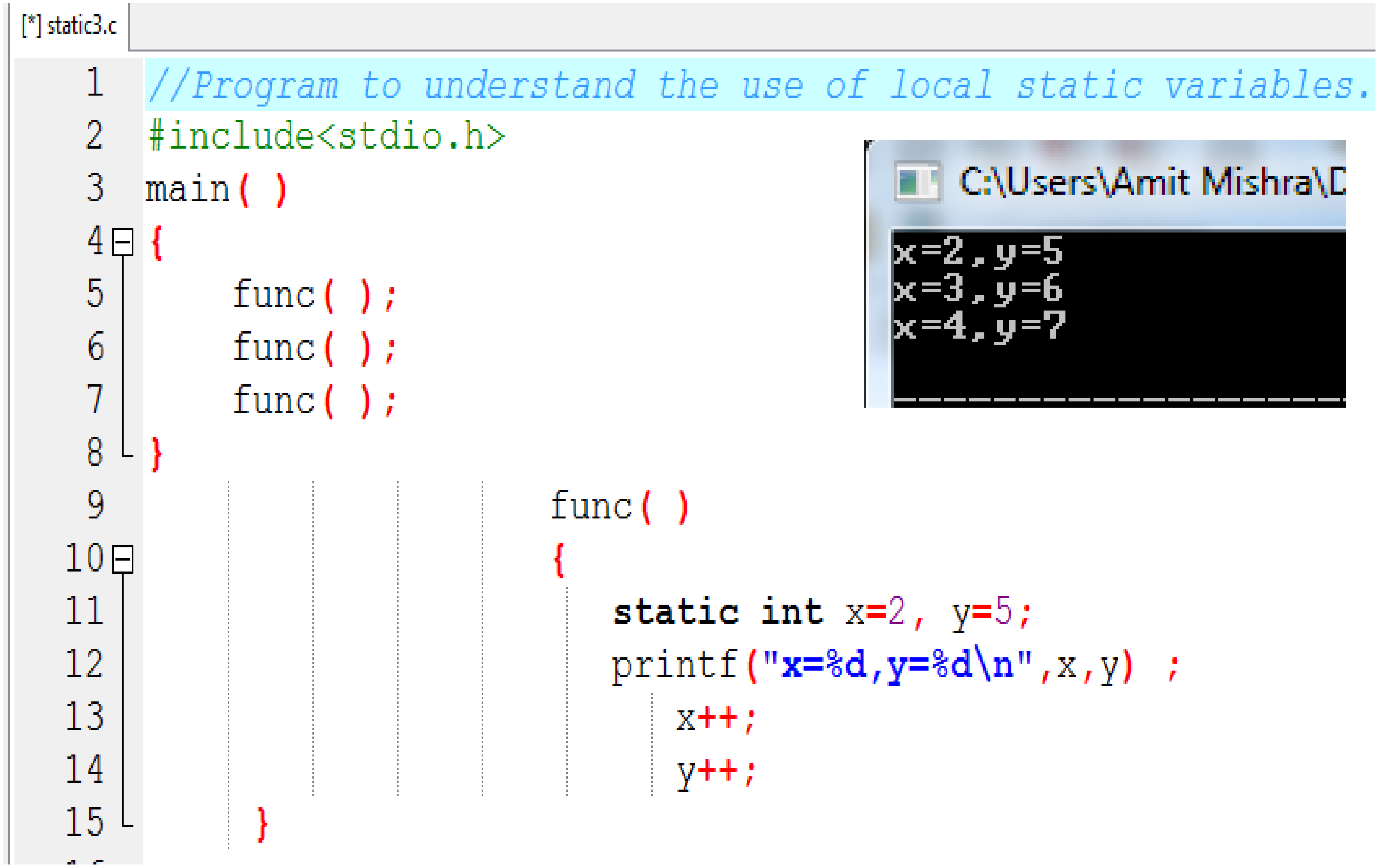
static2.cpp

```
//Program to demonstrate static.
#include<stdio.h>
void fun1();
main()
{
     fun1();
     fun1();
     fun1();

 }

     void fun1( )
          {
               static int x; // static variable
               x++;
               printf("\n x=%d \n",x);
          }
```

C:\Users\Amit Mishra\

```
x=1

x=2

x=3
```

- Static



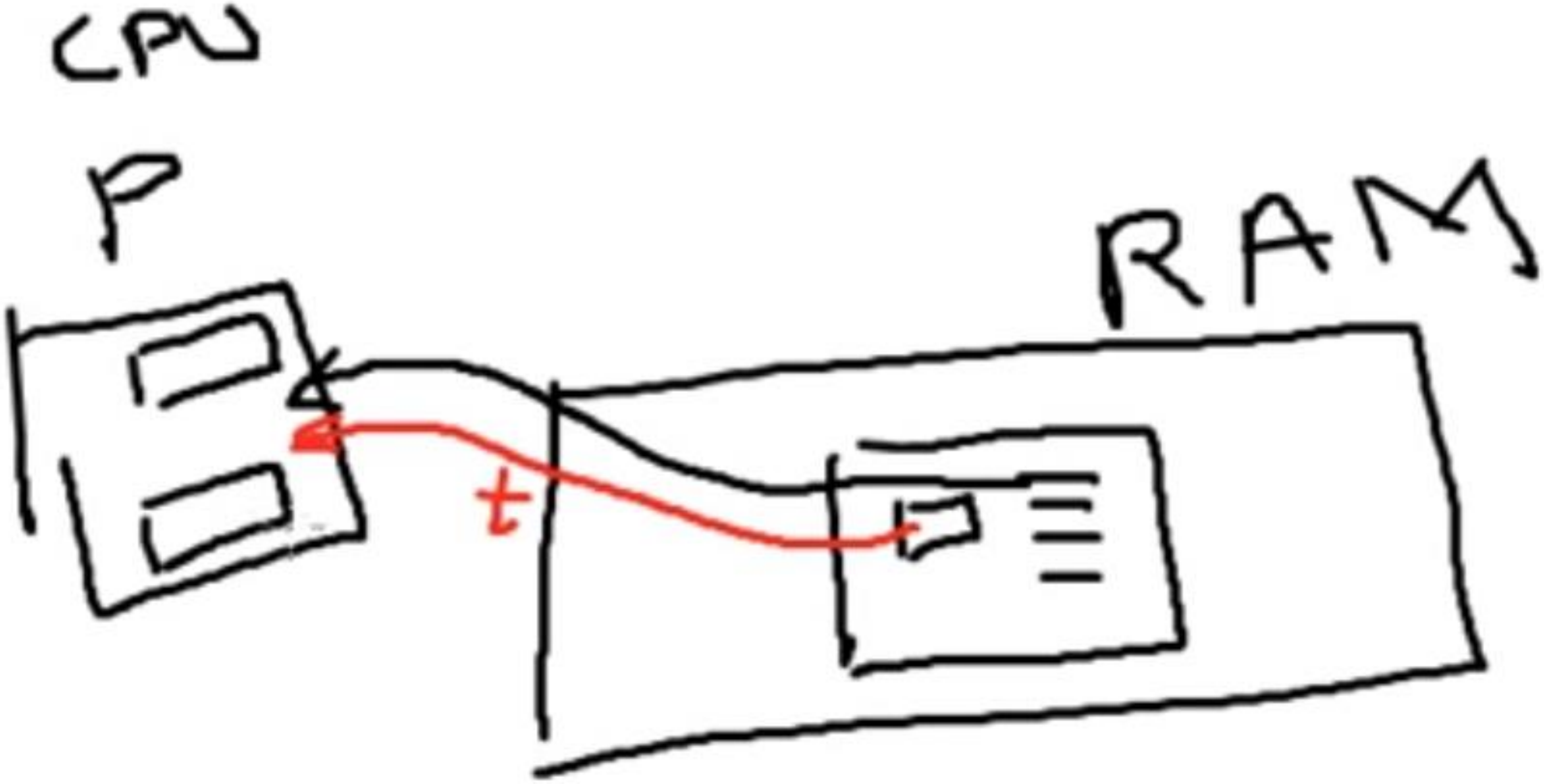
[*] static3.c

```c
//Program to understand the use of local static variables.
#include<stdio.h>
main( )
{
      func( );
      func( );
      func( );
}
                    func( )
                    {
                      static int x=2, y=5;
                      printf("x=%d,y=%d\n",x,y) ;
                          x++;
                          y++;
          }
```

C:\Users\Amit Mishra\D

```
x=2,y=5
x=3,y=6
x=4,y=7
```

- Register



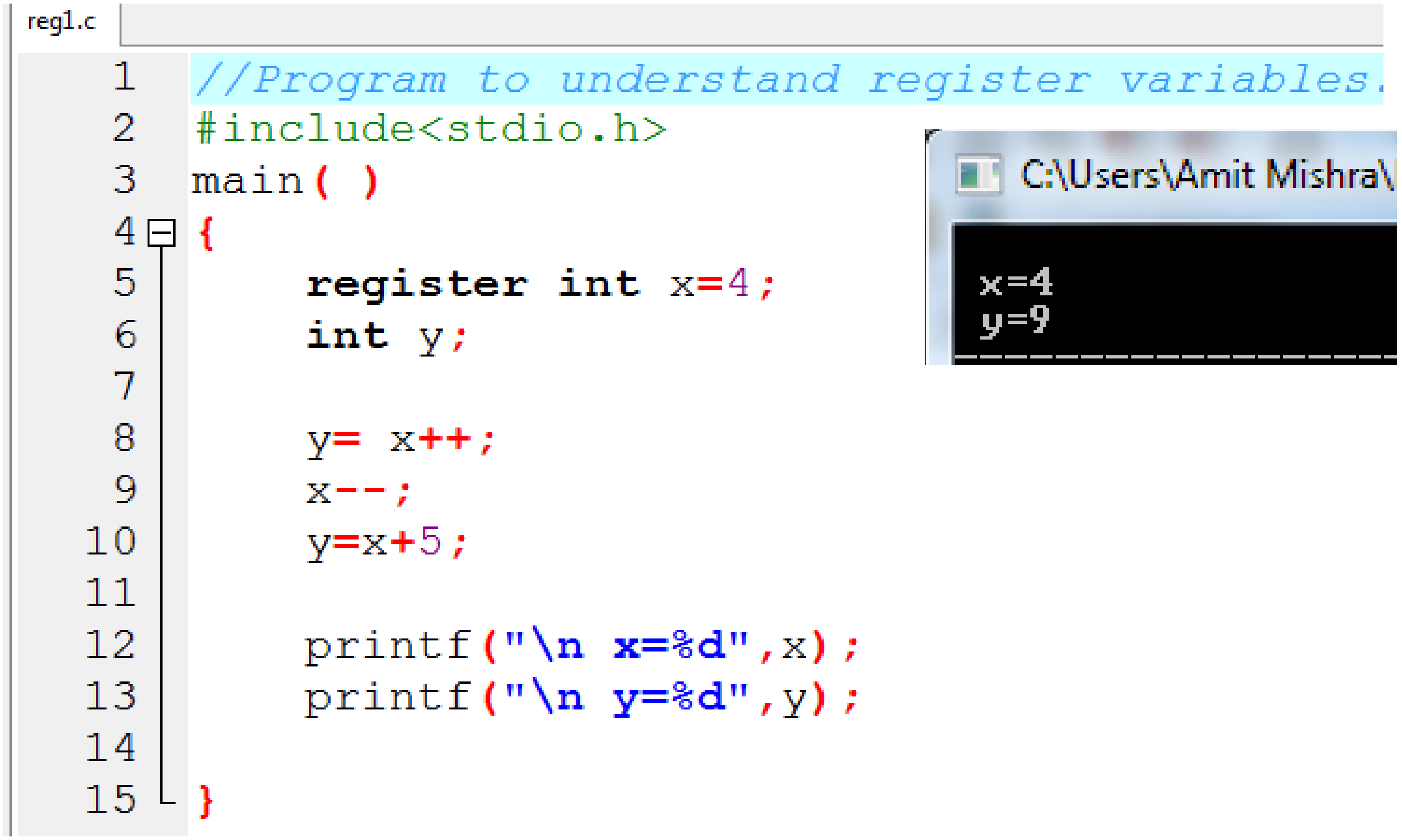
Register

- Register storage class can be applied only to local variables.
- The scope, lifetime and initial value of register variables are same as that of automatic variables.
- The only difference between the two is in the place where they are stored. Automatic variables are stored in memory while register variables are stored in CPU registers.
- Registers are small storage units present in the processor. The variables stored in registers can be accessed much faster than the variables stored in memory.
- So the variables that are frequently used can be assigned as register storage class for faster processing.

Register

- Register variables don't have memory addresses so we can't apply address operator(&) to them.
- There are limited number of registers in the processor hence we can declare only few variables as register.
- If many variables are declared as register and the CPU registers are not available then compiler will treat them as automatic variable.
- we can specify register storage class only for small size data type like int, char etc.
- If a variable other than these types is declared as register variable then compiler treat it as an automatic variable.

- Register



reg1.c

```
//Program to understand register variables.
#include<stdio.h>
main( )
{
	register int x=4;
	int y;

	y= x++;
	x--;
	y=x+5;

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

}
```

C:\Users\Amit Mishra\

```
x=4
y=9
```

FEATURE	STORAGE CLASS			
	Auto	Extern	Register	Static
Accessibility	Accessible within the function or block in which it is declared	Accessible within all program files that are a part of the program	Accessible within the function or block in which it is declared	Local: Accessible within the function or block in which it is declared Global: Accessible within the program in which it is declared
Storage	Main Memory	Main Memory	CPU Register	Main Memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Exists throughout the execution of the program	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared	Local: Retains value between function calls or block entries Global: Preserves value in program files
Default value	Garbage	Zero	Garbage	Zero