Functions in C

By: Atiya Jokhio

Functions

- Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters.
- Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or modules, each of which is more manageable than the original program.
- This technique is called divide and conquer.
- Modules in C are called functions.
- A larger program is created by developing smaller components individually and then assembling them together as whole.
- A function is a group of statements that together perform a task.
- Every C program has at least one function, which is main().

Types of Functions

- Depending on whether a function is defined by the user or already included in C compilers,
- There are two types of functions in C programming:
  - Standard library functions
  - User defined functions

Types of Functions [Cont.]

- **Standard library functions:**
- The C standard library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, and many other useful operations. These functions are also called as user defined functions.
- These functions are defined in the header file. When you include the header file, these functions are available for use. For example:
- The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file. There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

Types of Functions [Cont.]

- **User-defined functions**
- As mentioned earlier, C allows programmers to define functions.
- Depending upon the complexity and requirement of the program, we can create as many user-defined functions as we want.
- Because C does not provide every function that we will ever need, we must learn to write own functions.
- For example, there is no any function in C that finds the maximum number out of an array or swap two variables with each other.
- For these tasks we need to create our own (user-defined) functions.

Types of Functions [Cont.]

Example: User-defined function

Here is an example to add two integers. To perform this task, a user-defined function addNumbers() is defined.

```c
#include <stdio.h>

int addNumbers(int a, int b);       // function prototype

int main()
{ int n1,n2,sum;

    printf("Enters two numbers: "); scanf("%d
    %d",&n1,&n2);

    sum = addNumbers(n1, n2);       // function call

    printf("sum = %d",sum);

    return 0;
```

```c
int addNumbers(int a,int b)       // function definition
{ int result;
    result = a+b;
    return result;                // return statement
}
```

Why Functions

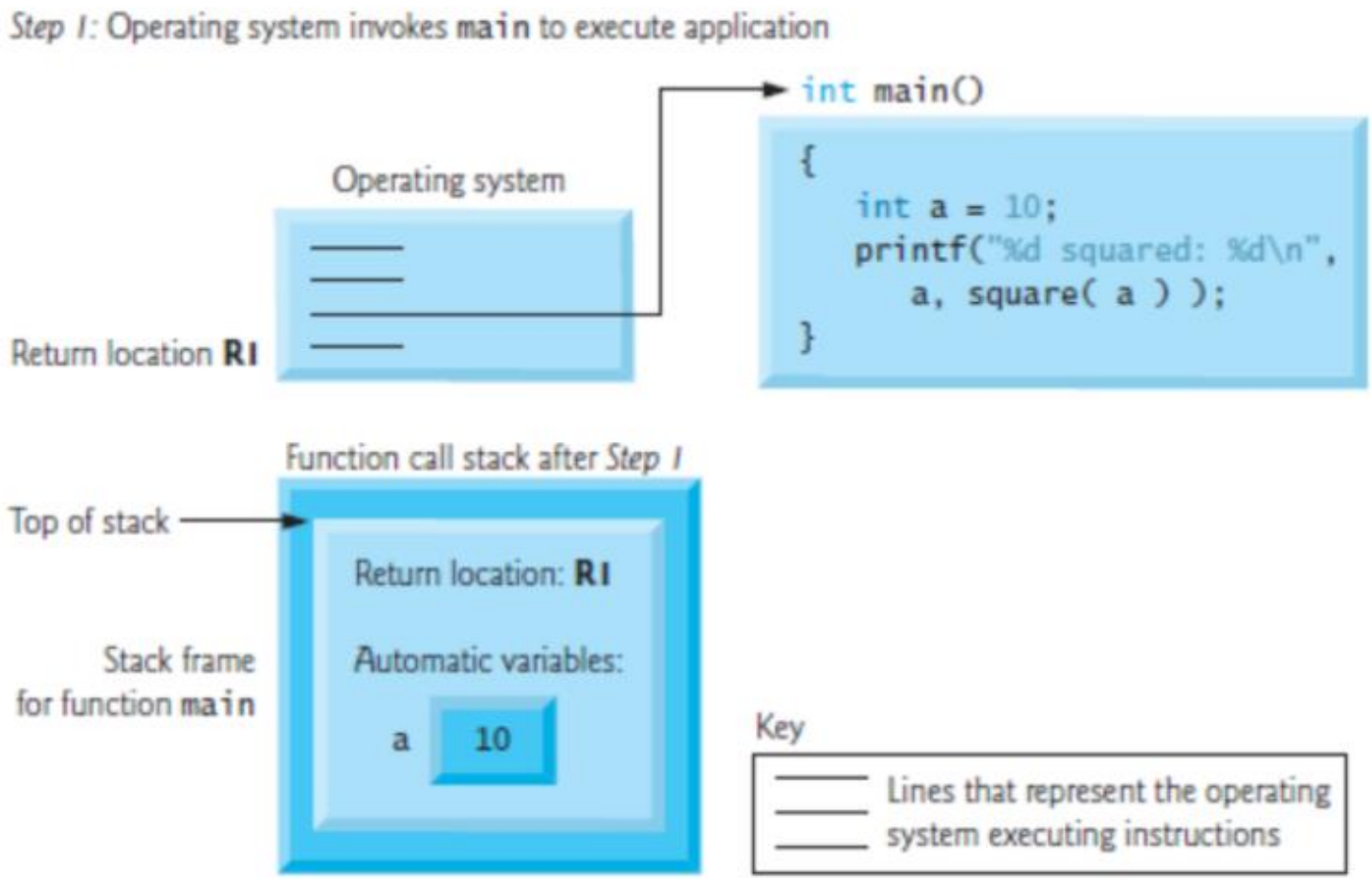
- We decompose a program into functions for several reasons:
- It provides modularity to your program's structure.
- It makes your code reusable. You just have to call the function by its name to use it, wherever required.
- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
- It makes the program more readable and easy to understand.

Function Call Stack and Stack Frames

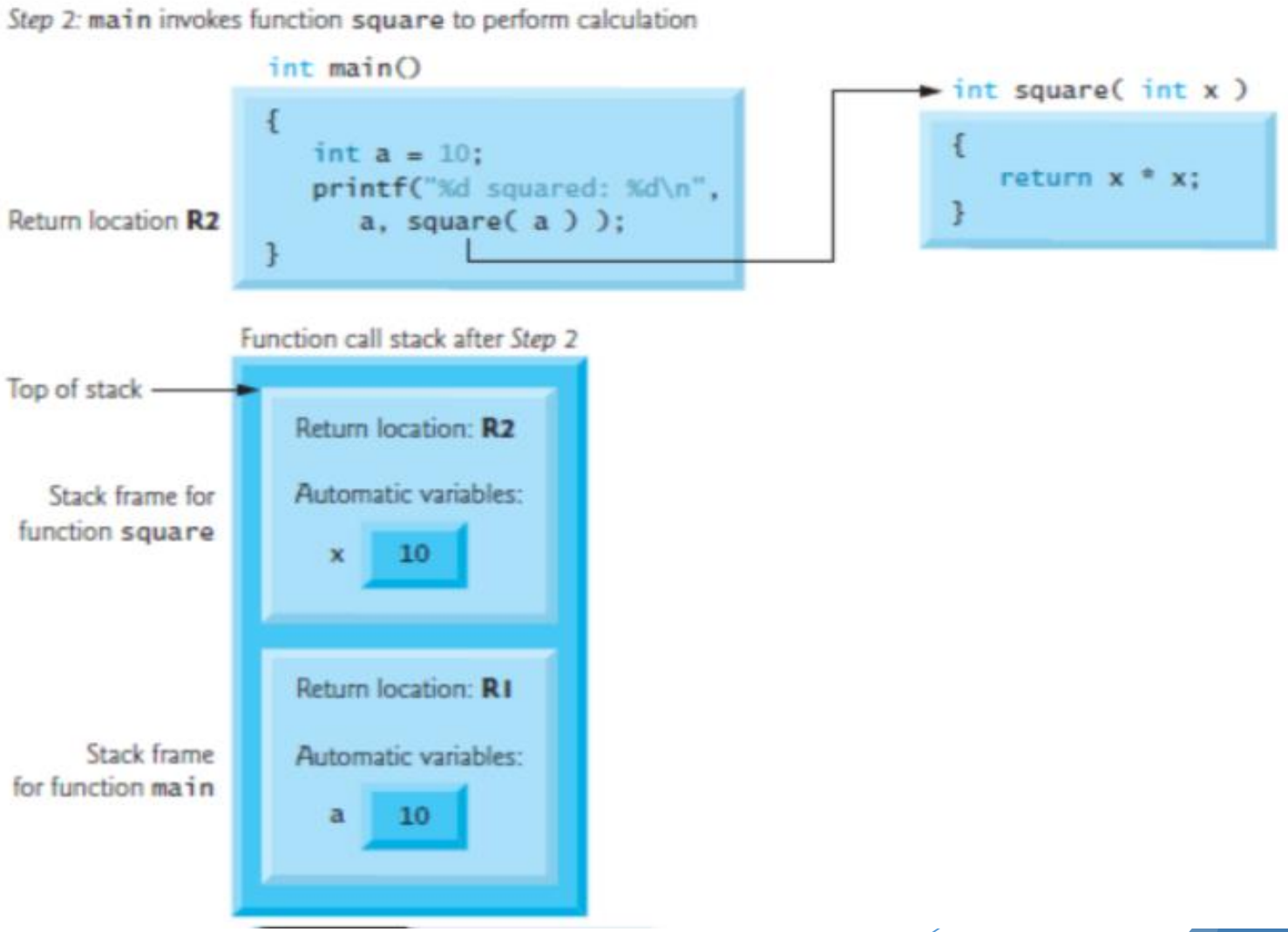
```
1  // Fig. 5.6: fig05_06.c
2  // Demonstrating the function call stack
3  // and stack frames using a function square.
4  #include <stdio.h>
5
6  int square( int ); // prototype for function square
7
8  int main()
9  {
10    int a = 10; // value to square (local automatic variable in main)
11
12    printf( "%d squared: %d\n", a, square( a ) ); // display a squared
13 } // end main
14
15 // returns the square of an integer
16 int square( int x ) // x is a local variable
17 {
18    return x * x; // calculate square and return result
19 } // end function square
```

```
10 squared: 100
```

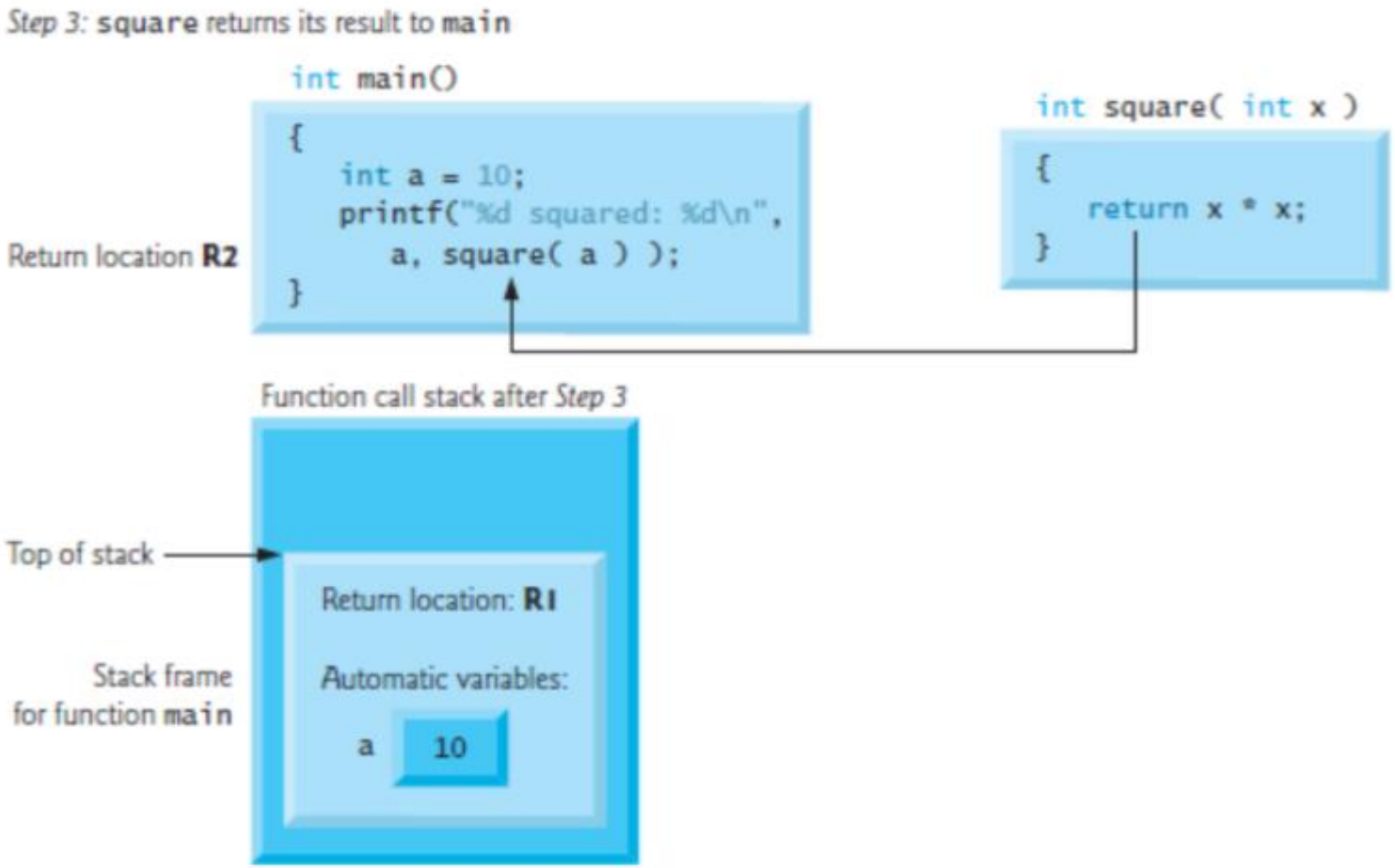
Function call stack after the operating system invokes main to execute the program.



Function call stack after main invokes square to perform the calculation.



Function call stack after function square returns to main.



Anatomy of Function

Function prototype

- A function prototype is simply the declaration of a function that specifies function's name, parameters and return type.
- It doesn't contain function body.
- A function prototype gives information to the compiler that the function may later be used in the program.

Anatomy of Function [Cont.]

Syntax of function prototype

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

In the above example, int addNumbers(int a, int b); is the function prototype which provides following information to the compiler:

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

Anatomy of Function [Cont.]

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

functionName(argument1, argument2, ...);

In the above example, function call is made using addNumbers(n1,n2); statement inside the main().

Function Calls

- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Function Calls

- Call by value

```
main( )
{
    int  a = 30 ;
    fun ( a ) ;
    printf ( "\n%d", a ) ;
}

fun ( int  b )
{
    b = 60 ;

  printf ( "\n%d", b ) ;
}
```

The output of the above program would be:

60
30

Anatomy of Function [Cont.]

Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

Syntax of function definition

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

Anatomy of Function [Cont.]

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Types:

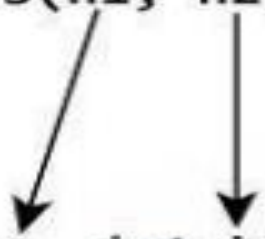
- No return and no parameter(s)
- No return but take parameter(s)
- Return some thing but no parameter(s)
- Return some thing as well as take parameter(s)

Function argument passing

- **Passing arguments to a function**
- In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during function call.
- The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

How to pass arguments to a function?

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```



Passing arguments

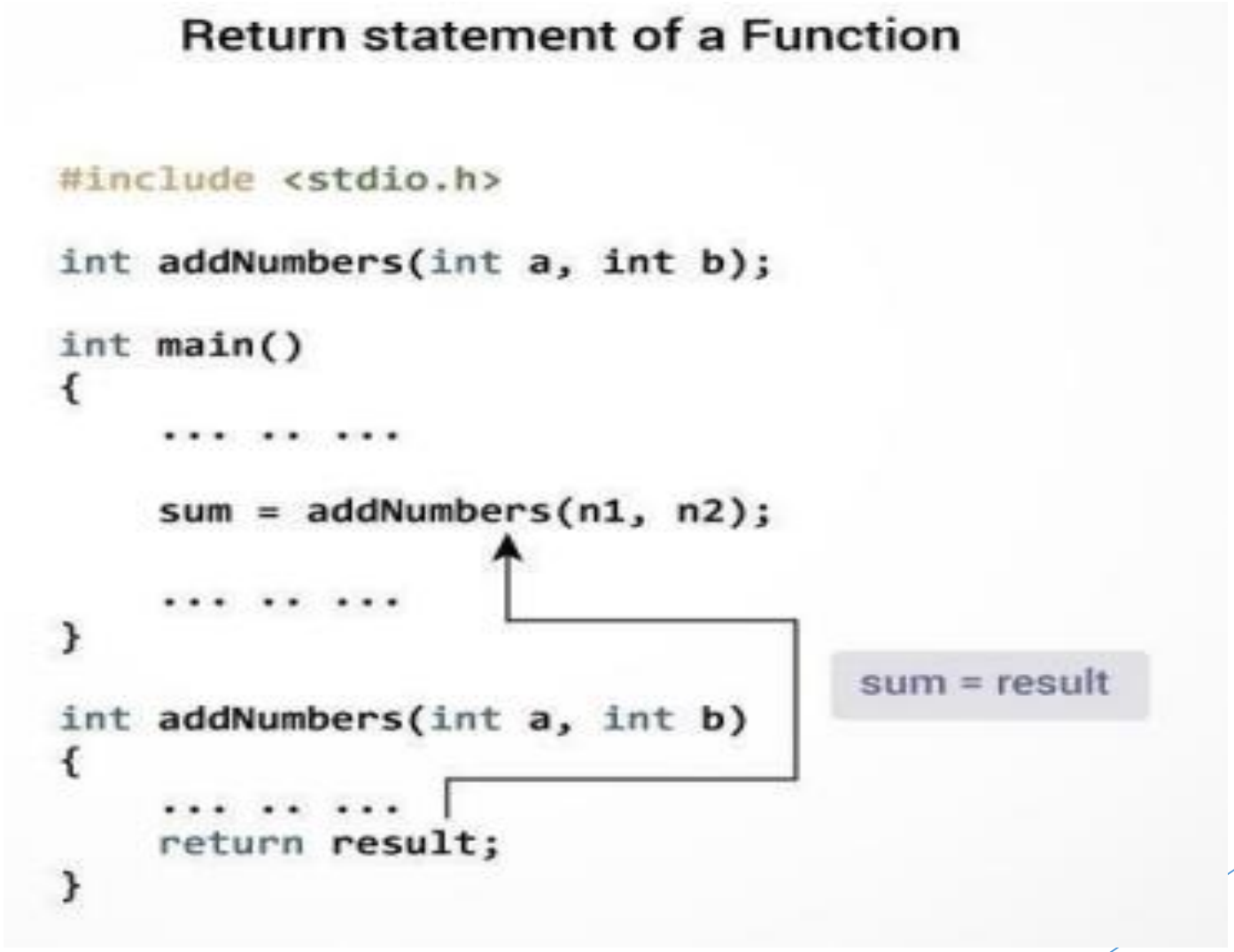
- The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.
- If n1 is of char type, a also should be of char type. If n2 is of float type, variable b also should be of float type.
- A function can also be called without passing an argument.

Return Statement

- The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.
- In the above example, the value of variable result is returned to the variable sum in the main() function.
- **Syntax of return statement**

```
return (expression);
```

Return Statement [Cont.]



Scope Rules

- The **scope of an identifier is the portion of the program in which the identifier can be referenced.**
- The four identifier scopes are:
  - function scope,
  - file scope,
  - block scope, and
  - function-prototype scope.
- Local variables declared as **static** *retain their values even when* they're out of scope.

What are the outputs?

```
void main(void)
{
        for (int i=1;i<=5;i++)
        {
            int c=1;
            c++;
            printf(" %d ", c);
        }
}
```

```
void main(void)
{
        for (int i=1;i<=5;i++)
        {
            static int c=1;
            c++;
            printf(" %d ", c);
        }
}
```

Storage class

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.
- **The static Storage Class**
- The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

Storage class

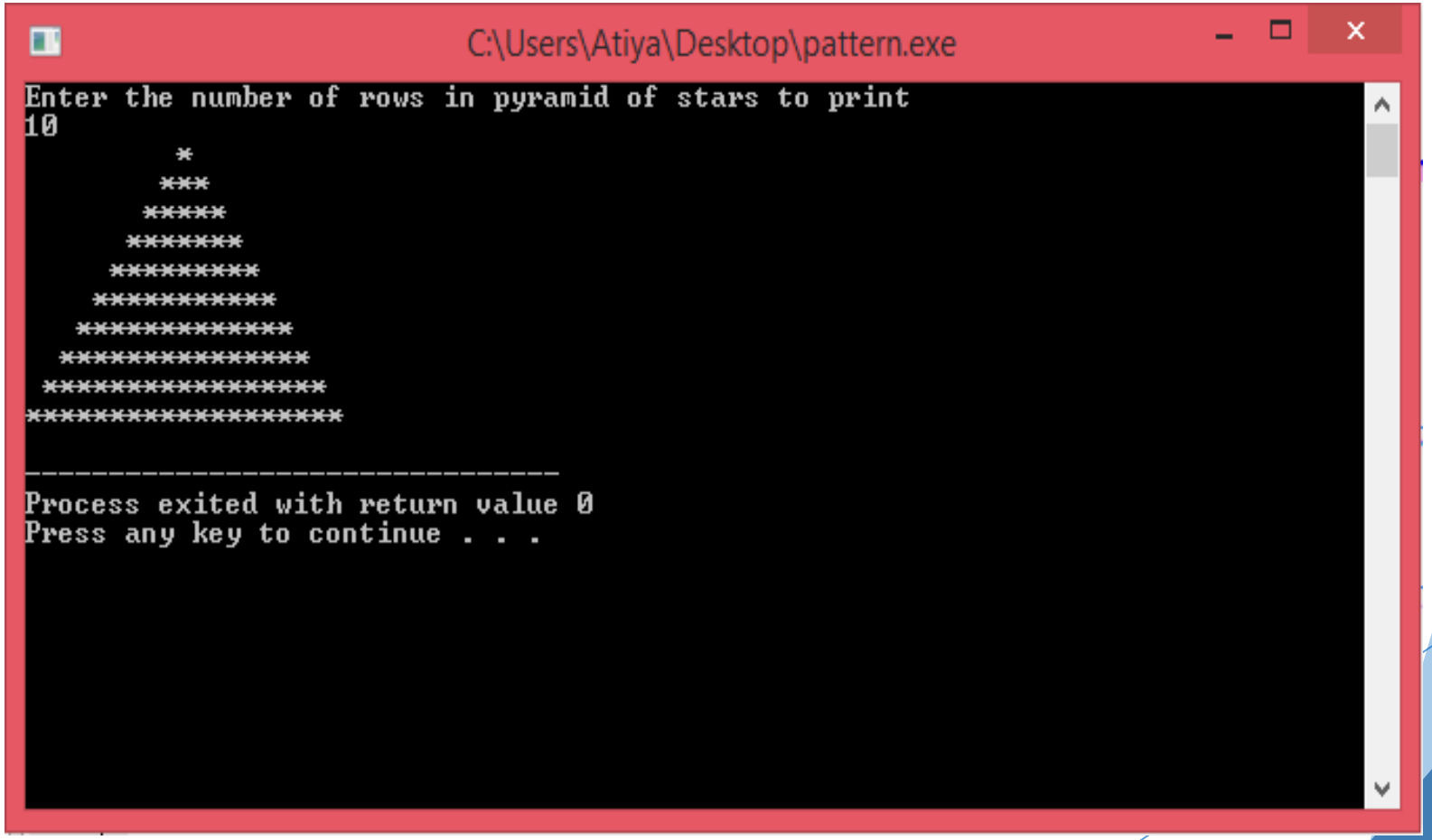
```c
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

Storage class

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

TASK:

Generate following pattern using nested loops and functions

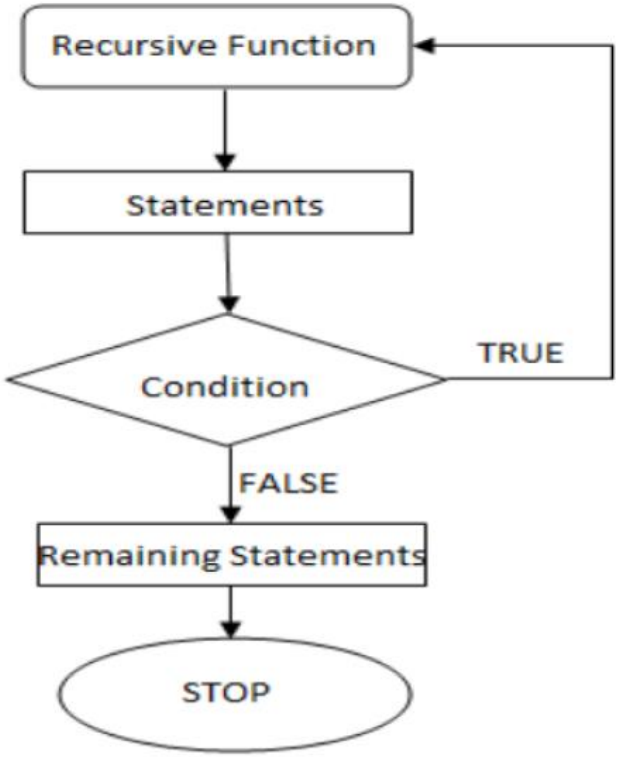


Recursion

- When a function invokes itself, the call is known as a recursive call.
- Recursion (the ability of a function to call itself) is an alternative control structure to repetition (looping). Rather than use a looping statement to execute a program segment, the program uses a selection statement to determine whether to repeat the code by calling the function again or to stop the process.

Recursion

Flowchart for recursion:



Recursion

- Each recursive solution has at least two cases: the base case and the general case.
- The **base case** is the one to which we have an answer;
- the **general case** expresses the solution in terms of a call to itself with a smaller version of the problem. Because the general case solves a smaller and smaller version of the original problem, eventually the program reaches the base case, where an answer is known, and the recursion stops.