

# 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

# 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.

# 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.

```
#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;
```

```
int addNumbers(int a,int b)    // function definition
```

```
{ int result;
```

```
    result = a+b;
```

```
    return result;    // return statement
```

```
}
```

# 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()`.

# 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.

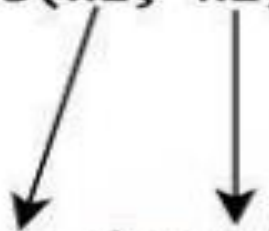
## How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

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



The diagram illustrates the flow of arguments. Two arrows originate from the variables `n1` and `n2` in the function call `addNumbers(n1, n2);` within the `main` function. These arrows point downwards to the parameters `a` and `b` in the function definition `int addNumbers(int a, int b)`, demonstrating how the values of `n1` and `n2` are passed to the function.

# 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.]

## Return statement of a Function

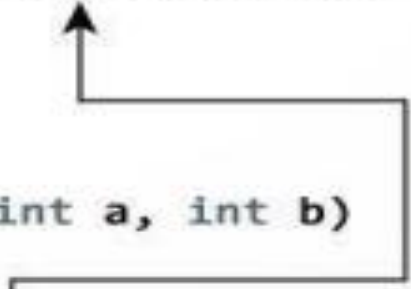
```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    return result;
}
```

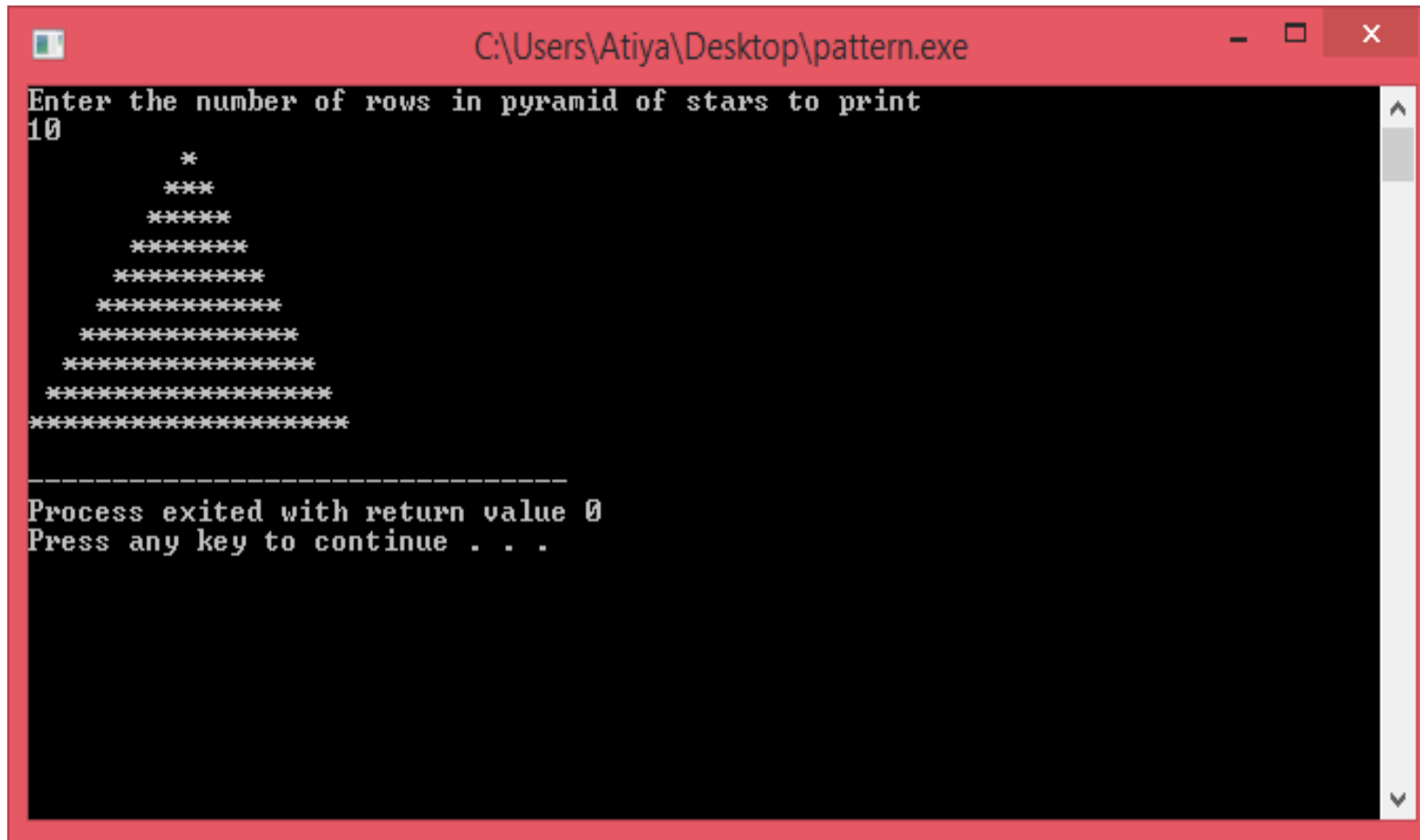
sum = result



The diagram illustrates the flow of data from the `addNumbers` function back to the `main` function. A line originates from the `return result;` statement in the `addNumbers` function, extends horizontally to the right, then turns vertically upwards, and finally turns horizontally to the left, ending with an arrowhead pointing to the `sum = addNumbers(n1, n2);` assignment in the `main` function. This visualizes how the value returned by the function is passed back to the caller.

## TASK:

Generate following pattern using nested loops and functions



```
C:\Users\Atiya\Desktop\pattern.exe
Enter the number of rows in pyramid of stars to print
10
      *
     ***
    *****
   *********
  ***********
 *****
*****
*****
*****
*****
*****
*****
*****
*****
*****
-----
Process exited with return value 0
Press any key to continue . . .
```