Functions in C

By Abdallah Ghazy

When the algorithm of a certain problem involves long and complex logic, it is broken into smaller, independent and reusable blocks.

These small blocks of code are known by different names in different programming languages such as a **module**, a **subroutine**, a **function** or a **method**.

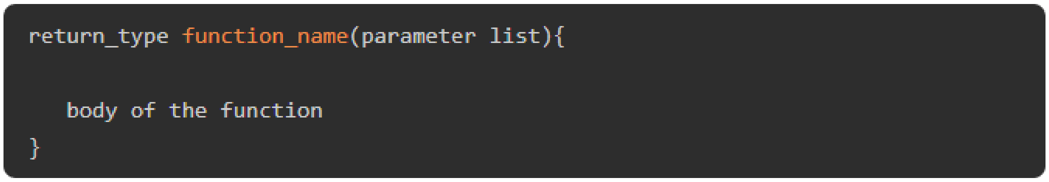
The main advantage of this approach is that the code becomes easy to follow, develop and maintain.

Library Functions in C

C offers a number of **library functions** included in different **header files**.

- ✓ **For example,** the **stdio.h** header file includes printf() and scanf() functions. Similarly, the **math.h** header file includes a number of functions such as sin(), pow(), sqrt() and more.

Parts of a Function in C



```
return_type function_name(parameter list){

   body of the function
}
```

- ✓ Return Type – A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

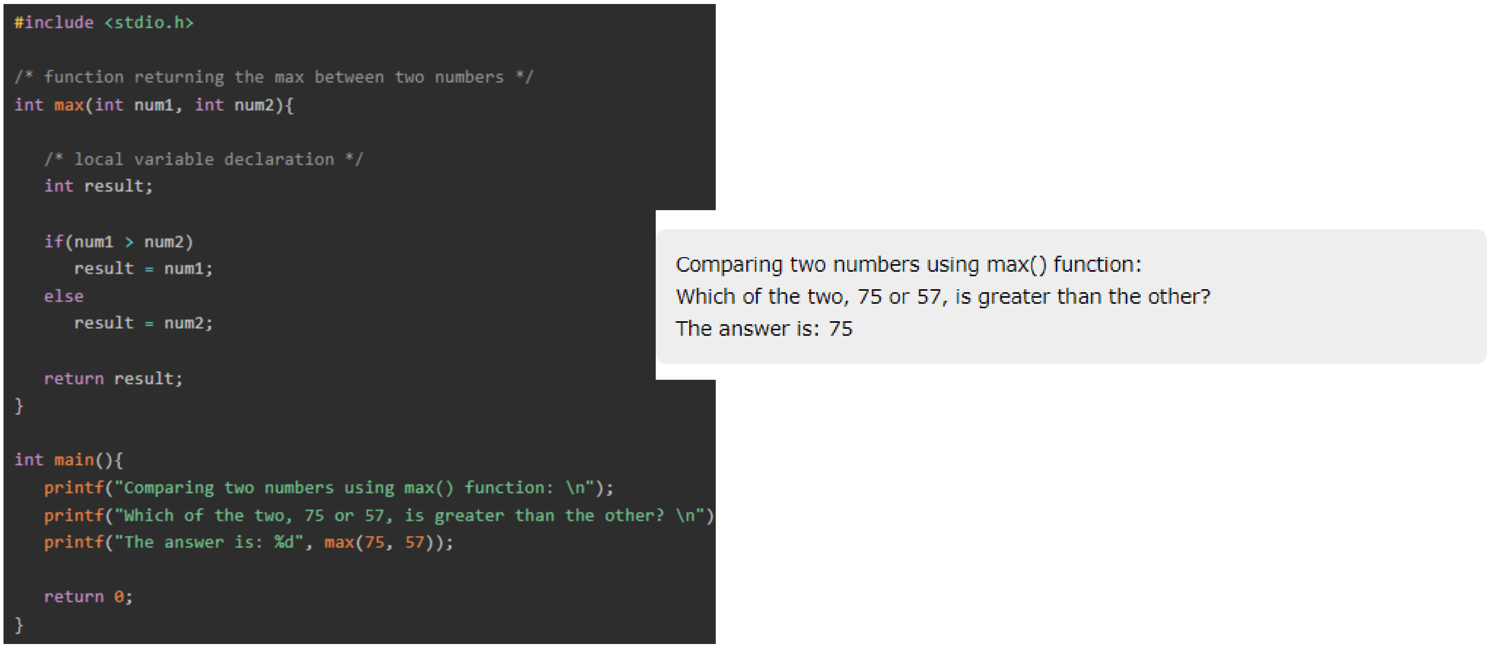
- ✓ Function Name – This is the actual name of the function.

- ✓ Argument List – An argument (also called *parameter*) is like a placeholder. When a function is invoked, you pass a value as a parameter. This value is referred to as the actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- ✓ Function Body – The function body contains a collection of statements that defines what the function does.

A function in C should have a return type. The type of the variable returned by the function must be the return type of the function. In the above figure, the add() function returns an int type.

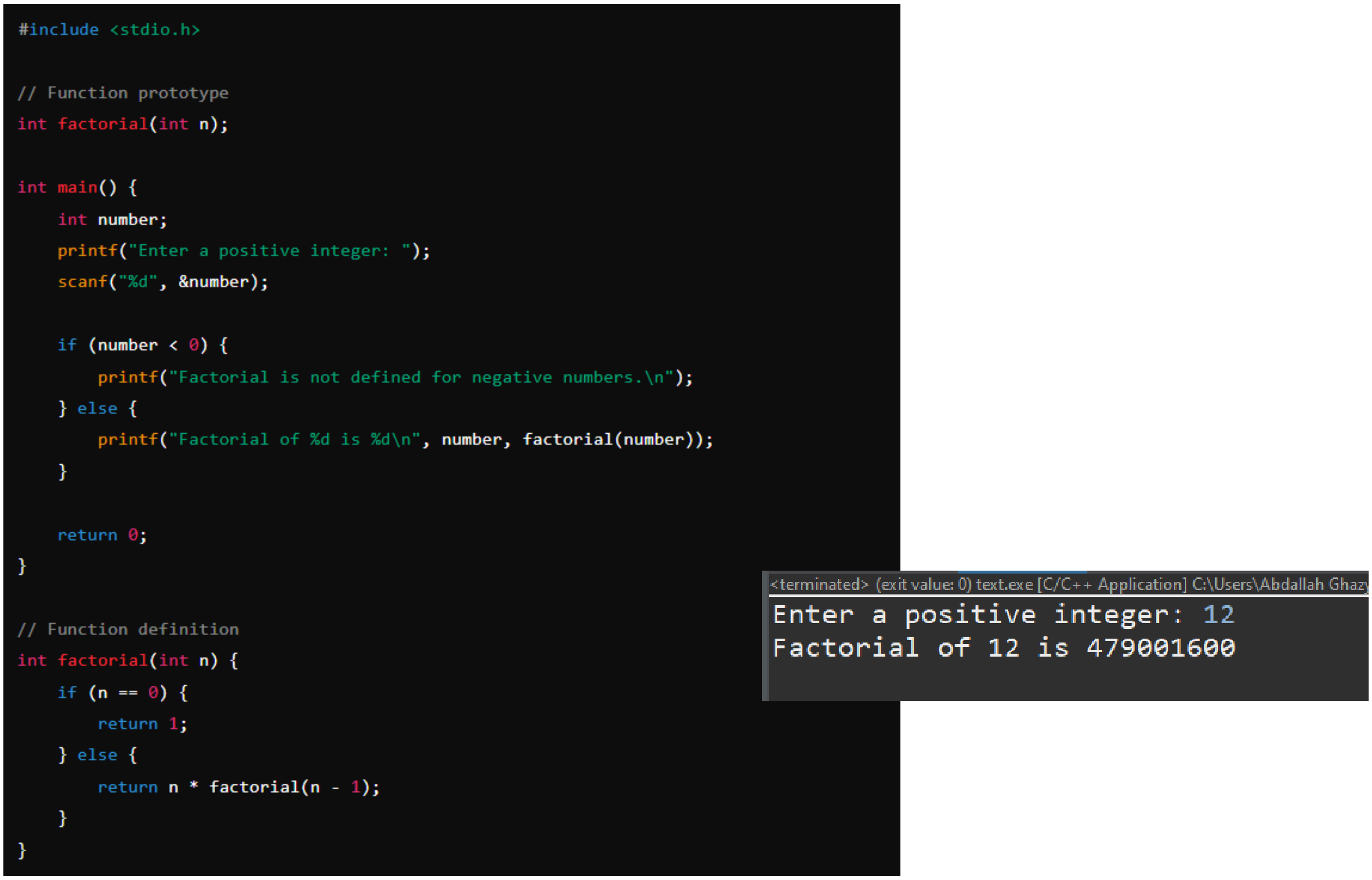
Example: User-defined Function in C



```c
#include <stdio.h>

/* function returning the max between two numbers */
int max(int num1, int num2){

   /* local variable declaration */
   int result;

   if(num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}

int main(){
   printf("Comparing two numbers using max() function: \n");
   printf("Which of the two, 75 or 57, is greater than the other? \n")
   printf("The answer is: %d", max(75, 57));

   return 0;
}
```

```
Comparing two numbers using max() function:
Which of the two, 75 or 57, is greater than the other?
The answer is: 75
```

Example: of a user-defined function in C. This function calculates the factorial of a given integer.



```c
#include <stdio.h>

// Function prototype
int factorial(int n);

int main() {
    int number;
    printf("Enter a positive integer: ");
    scanf("%d", &number);

    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        printf("Factorial of %d is %d\n", number, factorial(number));
    }

    return 0;
}

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

```
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy
Enter a positive integer: 12
Factorial of 12 is 479001600
```

Nested Functions in C

The term **nesting**, in programming context refers to enclosing a particular programming element inside another similar element. Just like nested loops, nested structures, etc., a nested function is a term used to describe the use of one or more functions inside another function.

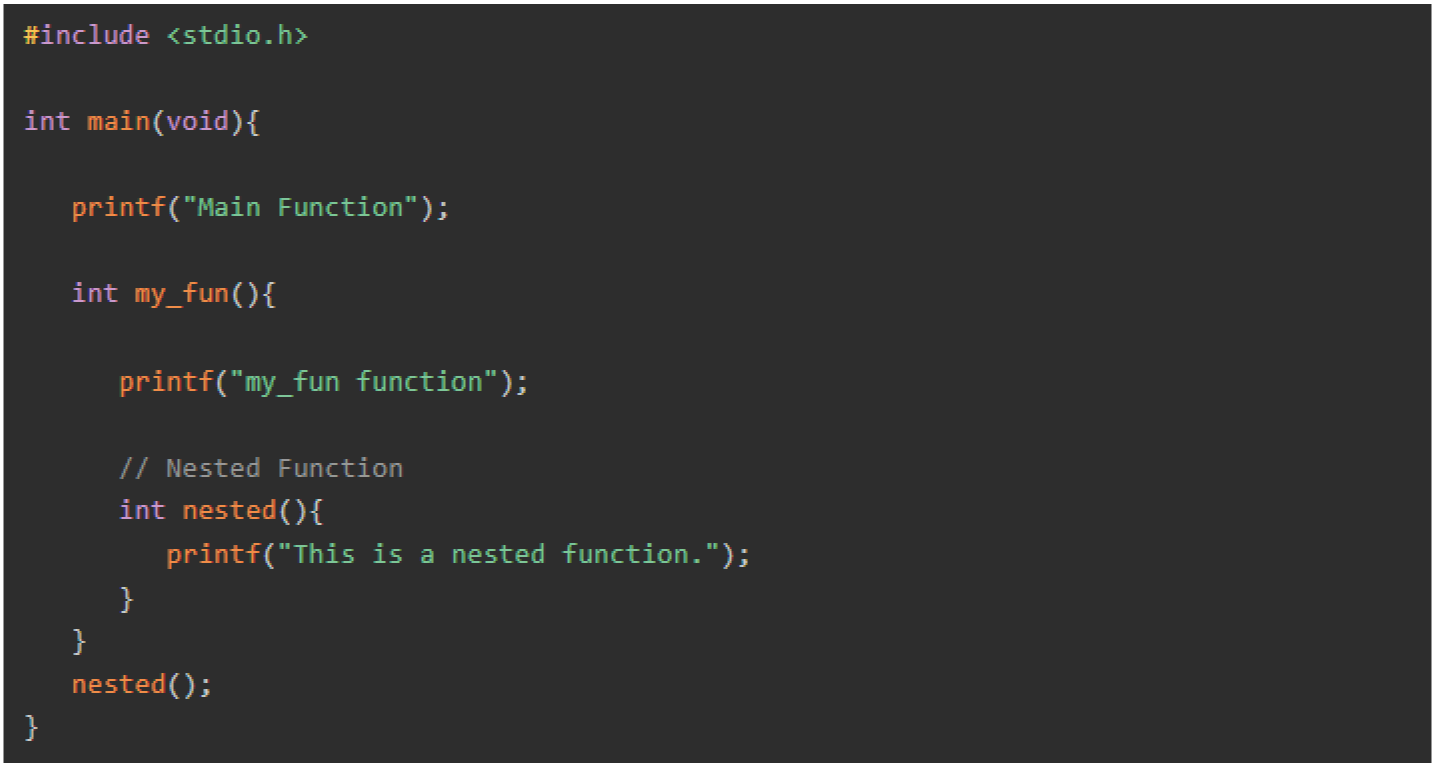
In C, the concept of nested functions (functions defined within other functions) is not supported by the standard C language. However, some compilers, such as GCC, provide extensions that allow for nested functions.

What is Lexical Scoping?

- In C language, defining a function inside another one is not possible.
- In short, nested functions are not supported in C. A function may only be **declared** (not **defined**) within another function.
- When a function is declared inside another function, it is called **lexical scoping**.
- Lexical scoping is not valid in C because the compiler cannot reach the correct memory location of inner function.

Nested Functions Have Limited Use

- Nested function definitions cannot access local variables of surrounding blocks.
- They can access only global variables. In C, there are two nested scopes: **local** and **global**.

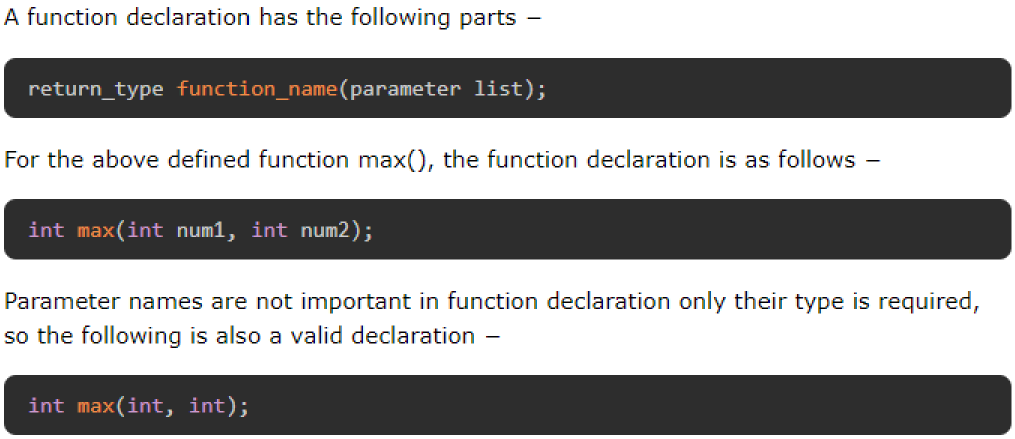


```c
#include <stdio.h>

int main(void){

   printf("Main Function");

   int my_fun(){

      printf("my_fun function");

      // Nested Function
      int nested(){
         printf("This is a nested function.");
      }
   }
   nested();
}
```

Declarations & definition & Calling

Function Declarations in C

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.



A function declaration has the following parts –

```
return_type function_name(parameter list);
```

For the above defined function max(), the function declaration is as follows –

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max(int, int);
```

Defining a Function in C

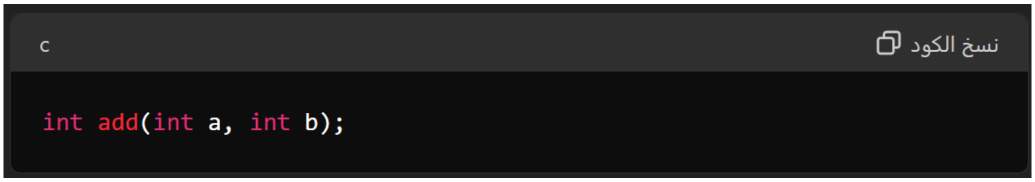
- ✓ In C, it is necessary to provide the forward declaration of the prototype of any function.
- ✓ The prototype of a library function is present in the corresponding header file.
- ✓ For a user-defined function, its prototype is present in the current program.
- ✓ The definition of a function and its prototype declaration should match.

Calling a Function in C

To call a function properly, you need to comply with the declaration of the function prototype. If the function is defined to receive a set of arguments, the same number and type of arguments must be passed.

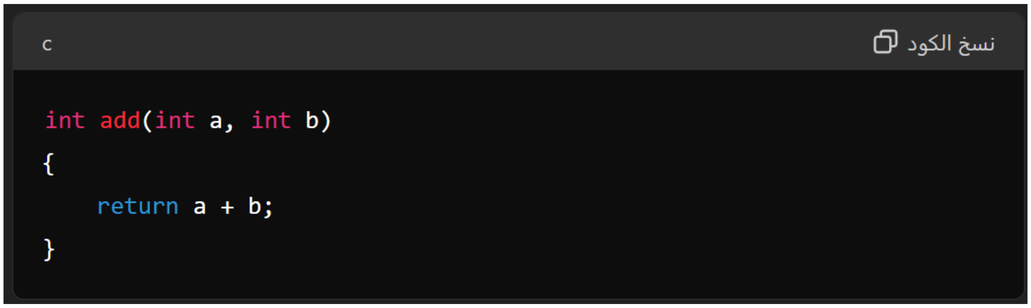
When a function is defined with arguments, the arguments in front of the function name are called **formal arguments**. When a function is called, the arguments passed to it are the **actual arguments**.

- ✓ **Function Declaration (or Prototype)**



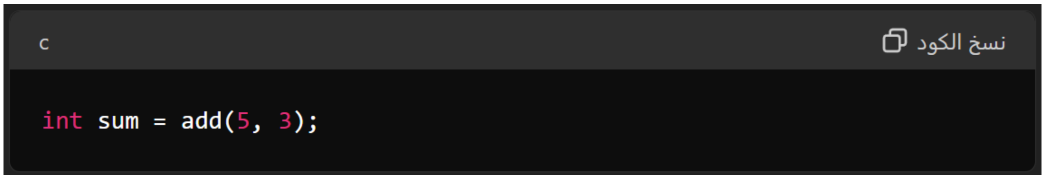
```c
int add(int a, int b);
```

- ✓ **Function Definition**



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

- ✓ **Function Call**



```c
int sum = add(5, 3);
```

Function Arguments

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

there are two ways in which arguments can be passed to a function –

Call by value (Pass by value)

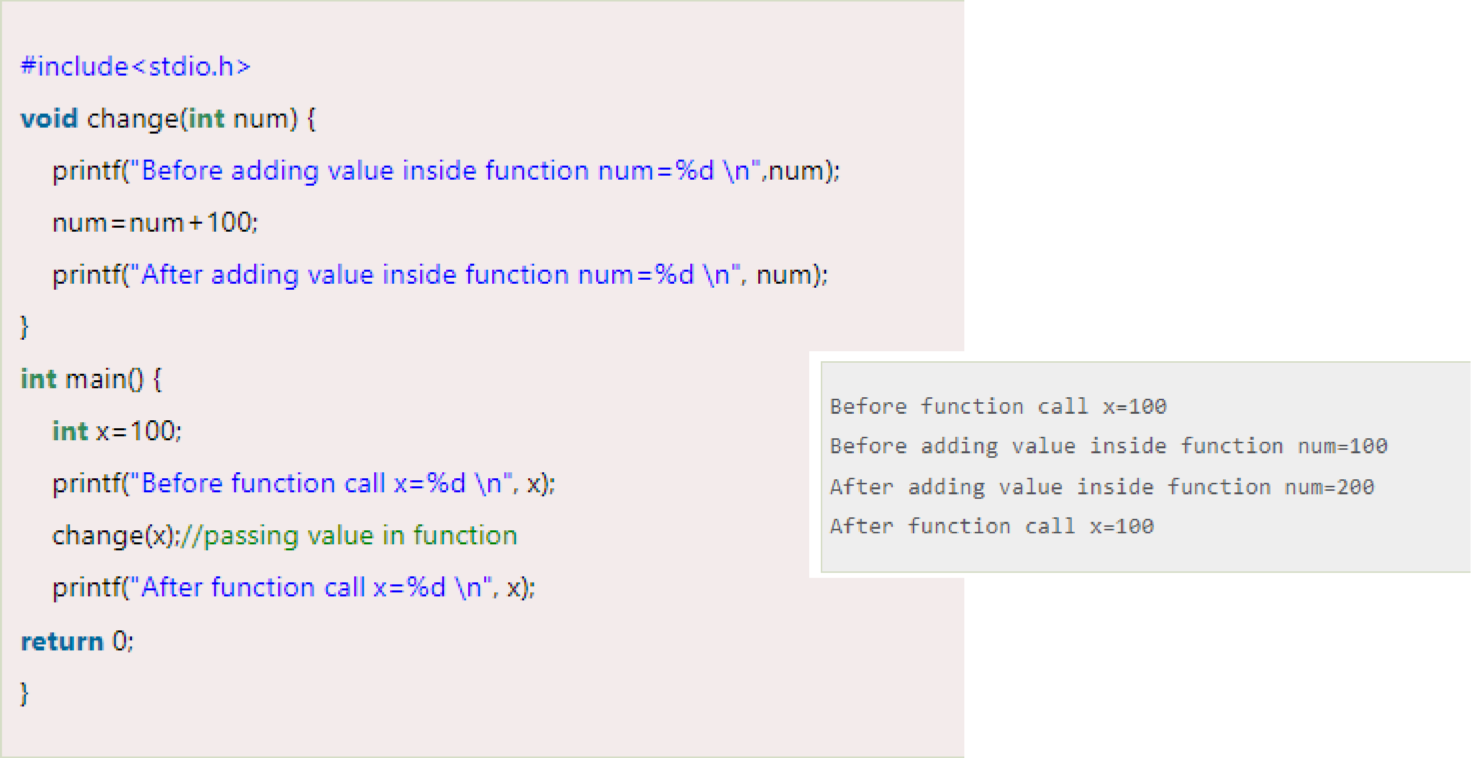
✓ This method copies the actual value of an argument into the formal parameter of the function.

✓

✓ In this case, changes made to the parameter inside the function have no effect on the argument.

✓ In call by value method, we cannot modify the value of the actual parameter by the formal parameter.

✓ In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

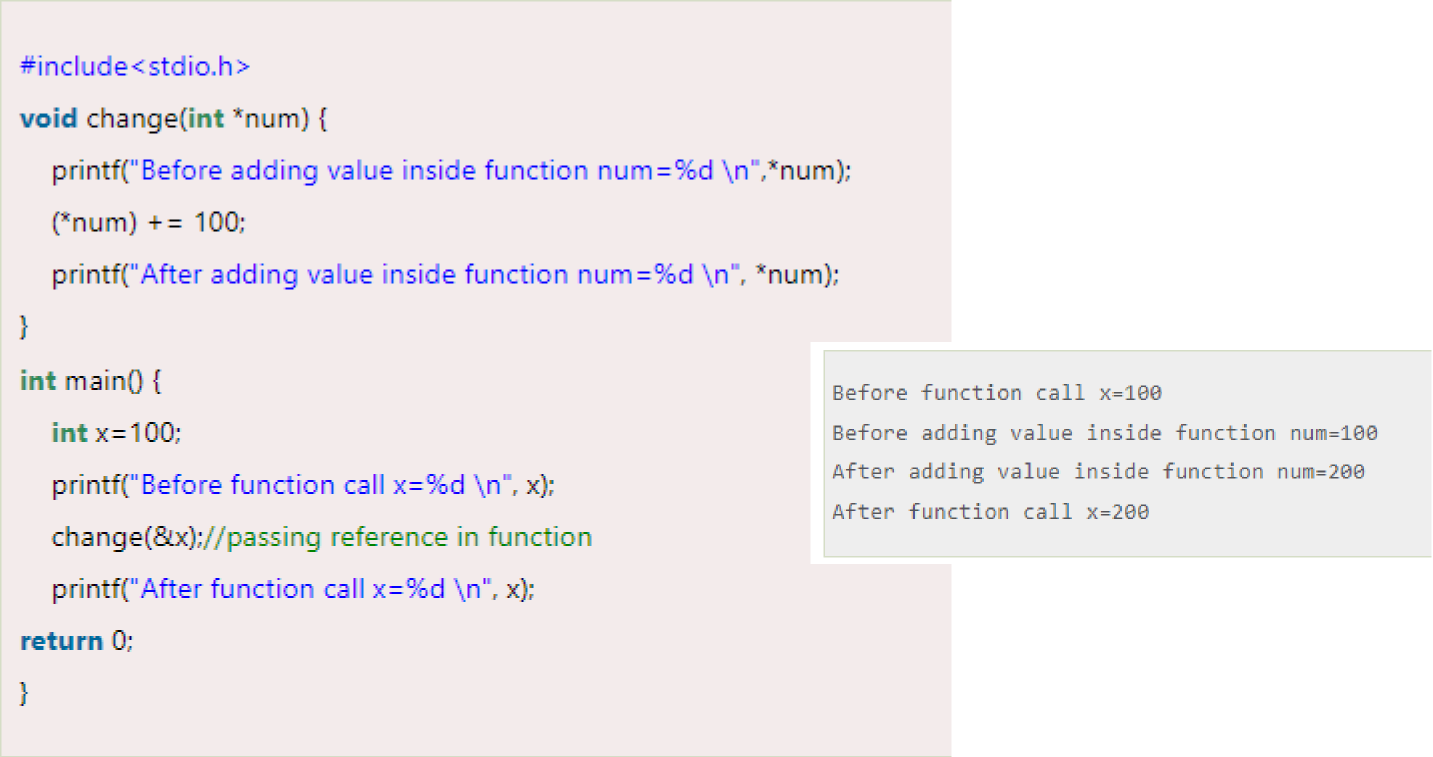


```
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by reference (Pass by reference)

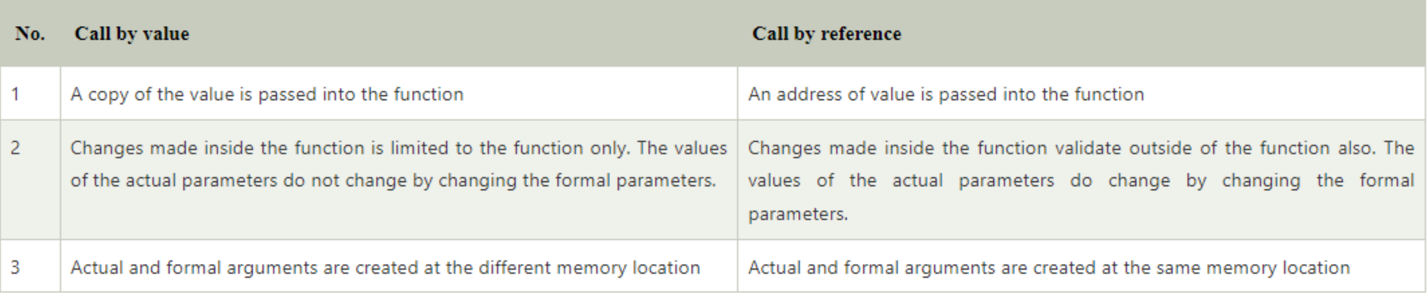
- ✓ This method copies the address of an argument into the formal parameter.
- ✓ Inside the function, the address is used to access the actual argument used in the call.
- ✓ This means that changes made to the parameter affect the argument.
- ✓ In call by reference, the memory allocation is similar for both formal parameters and actual parameters.
- ✓ All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.



```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
return 0;
}
```

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Difference between call by value and call by reference in c

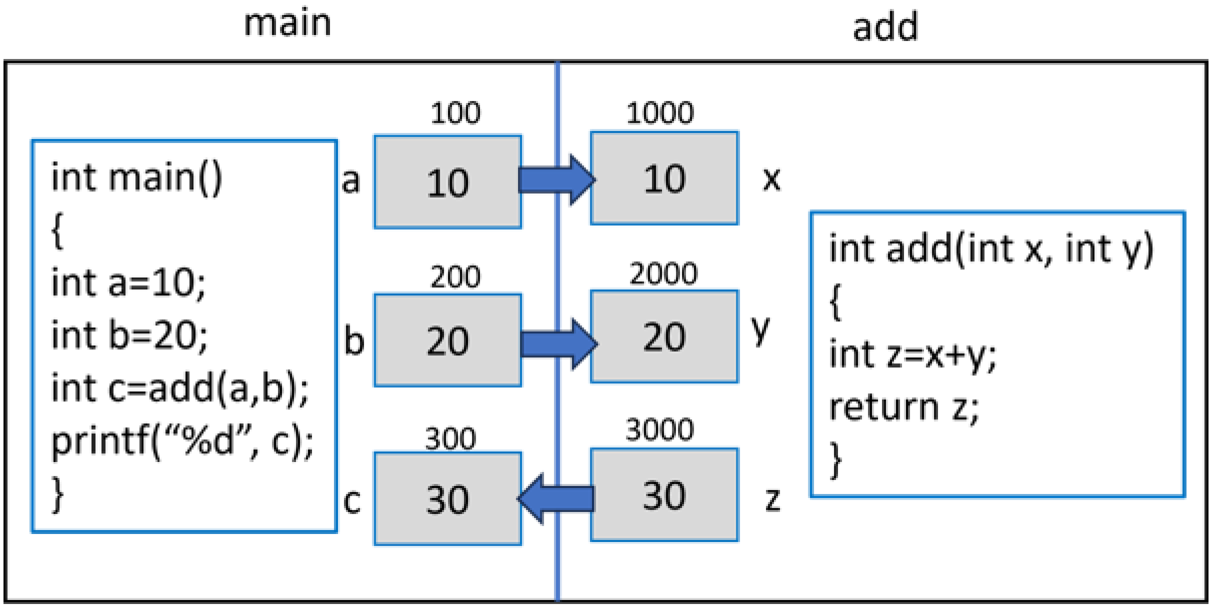


No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

How Does "Call by Value" Work in C?

Let us assume that the variables **a**, **b** and **c** in the main() function occupy the memory locations 100, 200 and 300 respectively.

When the add() function is called with **a** and **b** as actual arguments, their values are stored in **x** and **y** respectively.



- ✓ The variables **x**, **y**, and **z** are the local variables of the add() function.
- ✓ In the memory, they will be assigned some random location.
- ✓ Let's assume that they are created in memory address 1000, 2000 and 3000, respectively.

Since the function is called by copying the value of the actual arguments to their corresponding formal argument variables, the locations 1000 and 2000 which are the address of **x** and **y** will hold 10 and 20, respectively.

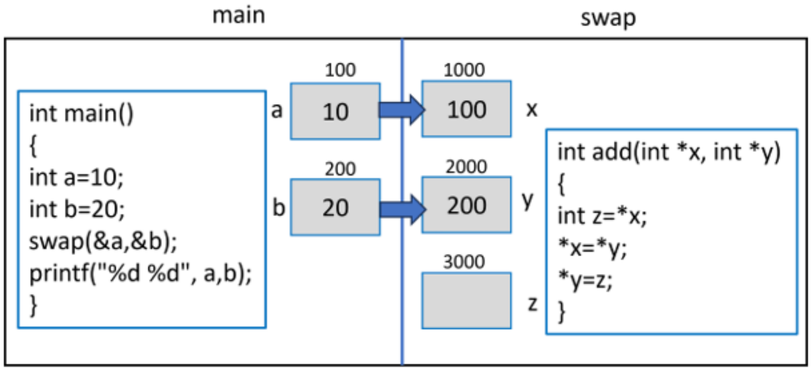
The compiler assigns their addition to the third local variable **z** which is returned.

As the control comes back to the main() function, the returned data is assigned to **c**, which is displayed as the output of the program.

How Does "Call by reference" Work in C?

Assume that the variables **a** and **b** in the main() function are allotted locations with the memory address 100 and 200 respectively.

As their addresses are passed to **x** and **y** (remember that they are pointers), the variables **x**, **y** and **z** in the swap() function are created at addresses 1000, 2000 and 3000 respectively.



```
int main()
{
int a=10;
int b=20;
swap(&a,&b);
printf("%d %d", a,b);
}
```

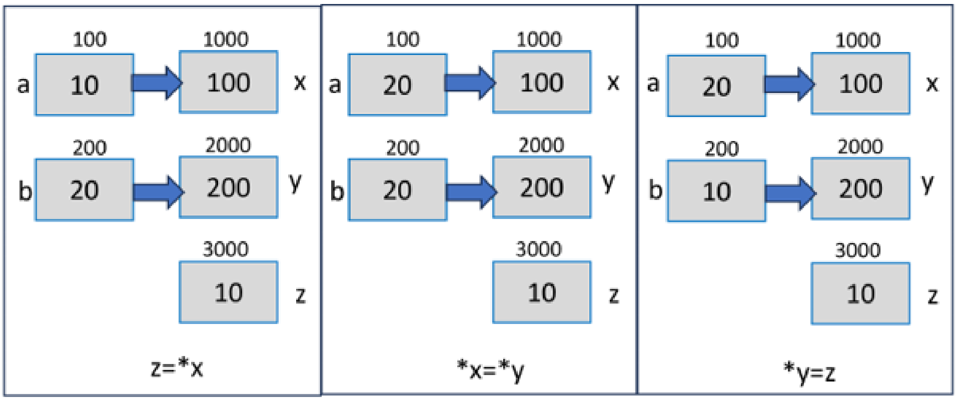
```
int add(int *x, int *y)
{
int z=*x;
*x=*y;
*y=z;
}
```

Since "x" and "y" store the address of "a" and "b", "x" becomes 100 and "y" becomes 200, as the above figure shows.

Inside the swap() function, the first statement "**z = *x**" causes the value at address in "x" to be stored in "x" (which is 10).

Similarly, in the statement "***x = *y;**", the value at the address in "y" (which is 20) is stored in the location whose pointer is "x".

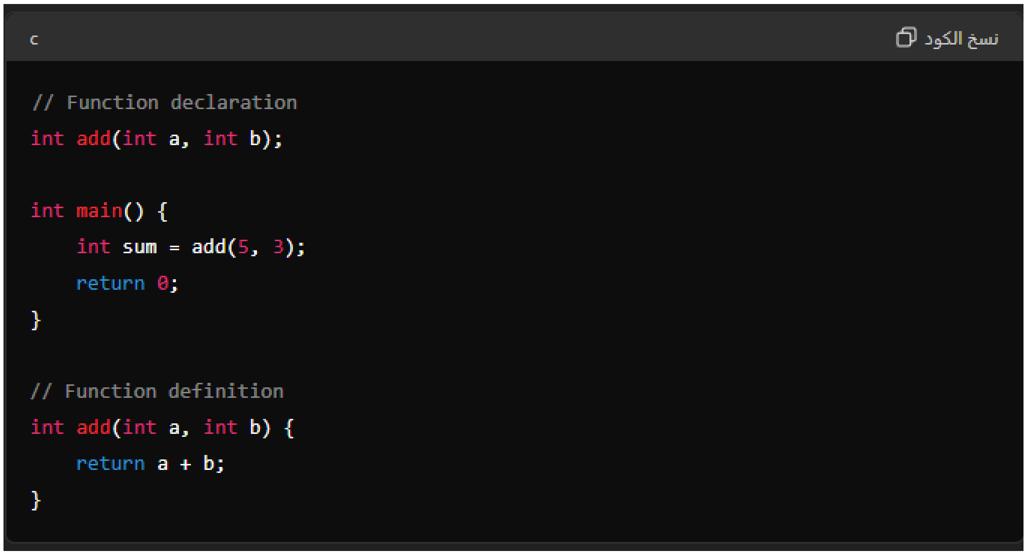
Finally, the statement "***y = z;**" assigns the "z" to the variable pointed to by "y", which is "b" in the main() function. The values of "a" and "b" now get swapped.



the potential errors when using functions in C

Not Declaring the Function Before Use:

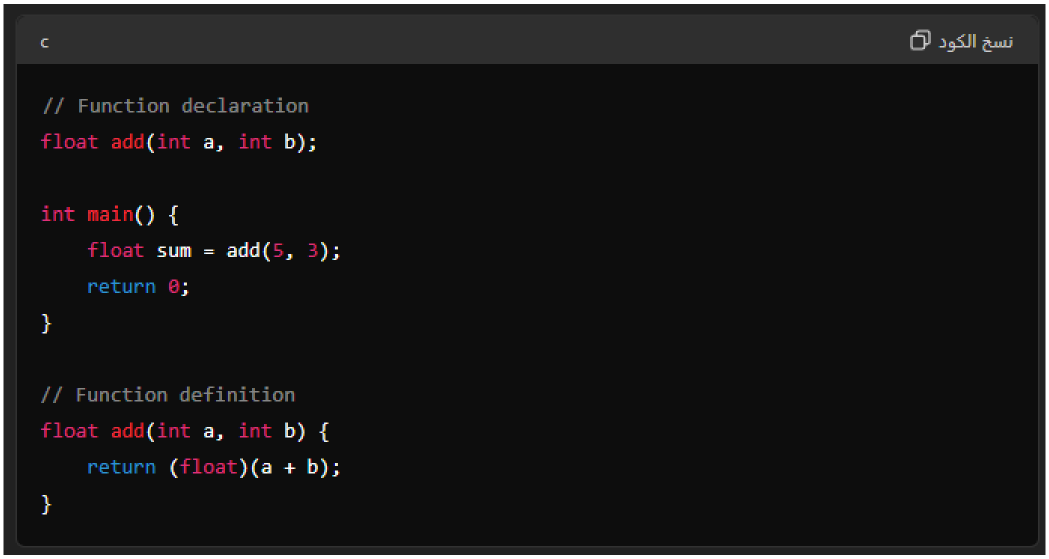
- **Error:** If you don't declare the function before using it, a compile-time error may occur.
- **Solution:** Ensure to declare the function using a declaration statement before using it.



```c
// Function declaration
int add(int a, int b);

int main() {
    int sum = add(5, 3);
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

Mismatch in Return Type or Parameters Between Declaration and Definition:

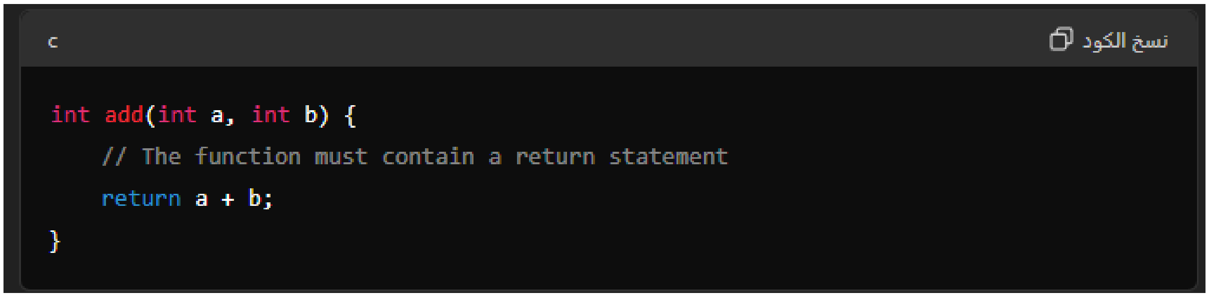
- **Error:** If the return type or parameter types in the function definition differ from those in the declaration, errors will occur.
- **Solution:** Ensure that the return type and parameters in the declaration and definition match.



```c
// Function declaration
float add(int a, int b);

int main() {
    float sum = add(5, 3);
    return 0;
}

// Function definition
float add(int a, int b) {
    return (float)(a + b);
}
```

Forgetting to Return a Value in the Function:

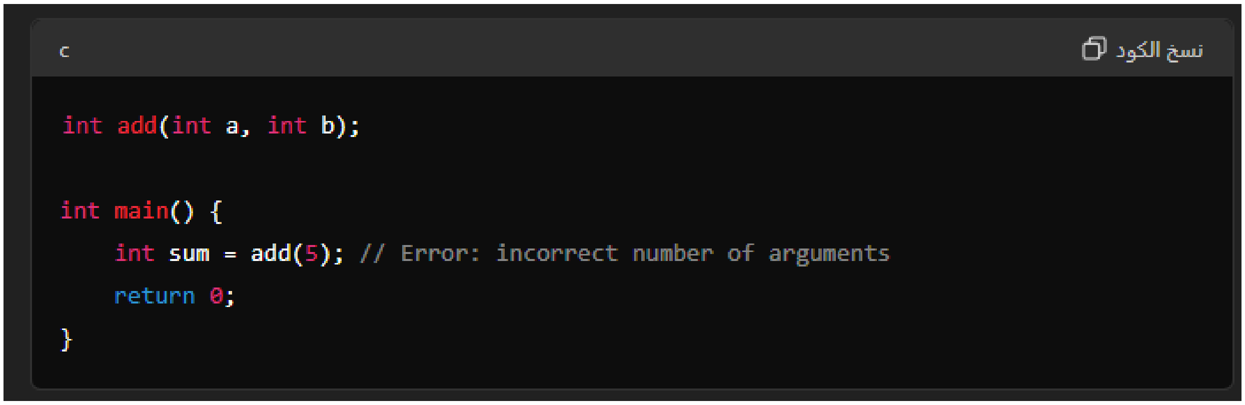
- **Error:** If the function requires a return value (e.g., int) and you don't return a value in the function definition, errors will occur.
- **Solution:** Ensure to use the return statement to return an appropriate value.



```c
int add(int a, int b) {
    // The function must contain a return statement
    return a + b;
}
```

Sending the Wrong Number of Arguments When Calling the Function:

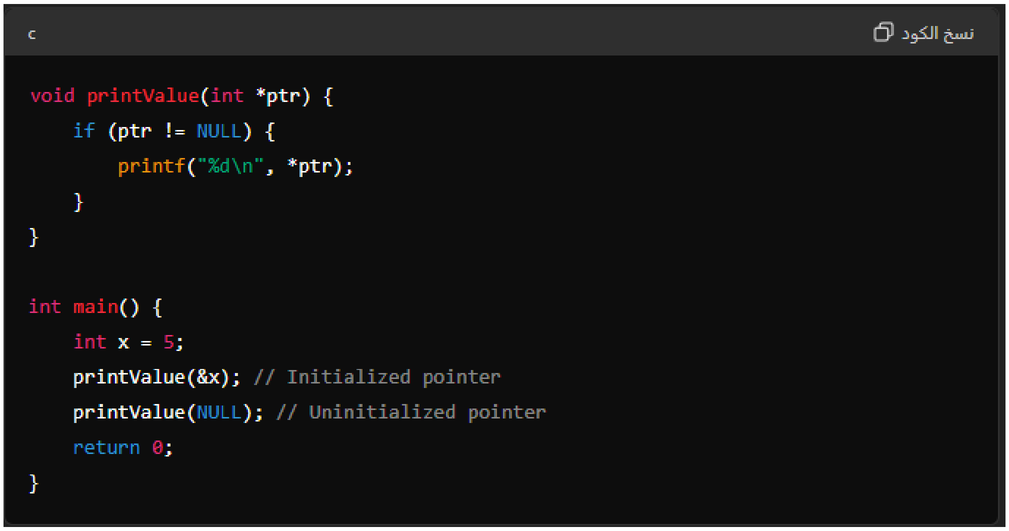
- **Error:** If you send the wrong number of arguments when calling the function, errors will occur.
- **Solution:** Ensure that the number and types of arguments sent match the function definition.



```c
int add(int a, int b);

int main() {
    int sum = add(5); // Error: incorrect number of arguments
    return 0;
}
```

Using an Uninitialized Pointer Inside the Function:

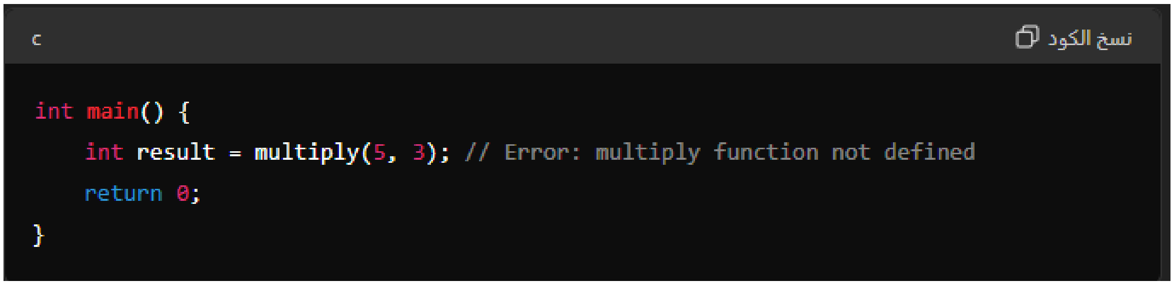
- **Error:** If you use an uninitialized pointer inside the function, runtime errors or program crashes may occur.
- **Solution:** Ensure to initialize pointers before using them.



```c
void printValue(int *ptr) {
    if (ptr != NULL) {
        printf("%d\n", *ptr);
    }
}

int main() {
    int x = 5;
    printValue(&x); // Initialized pointer
    printValue(NULL); // Uninitialized pointer
    return 0;
}
```

Using a Function That Hasn't Been Defined:

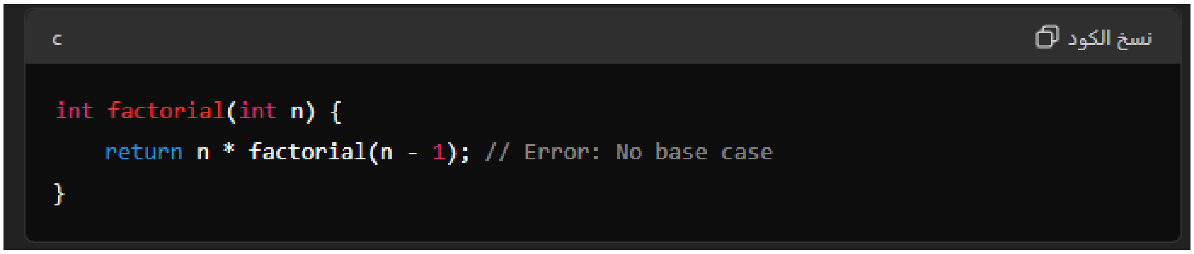
- **Error:** If you call a function that hasn't been defined, the linker will generate an error.
- **Solution:** Ensure that all functions used in the program are defined.



```c
int main() {
    int result = multiply(5, 3); // Error: multiply function not defined
    return 0;
}
```

Mismatched Return Type:

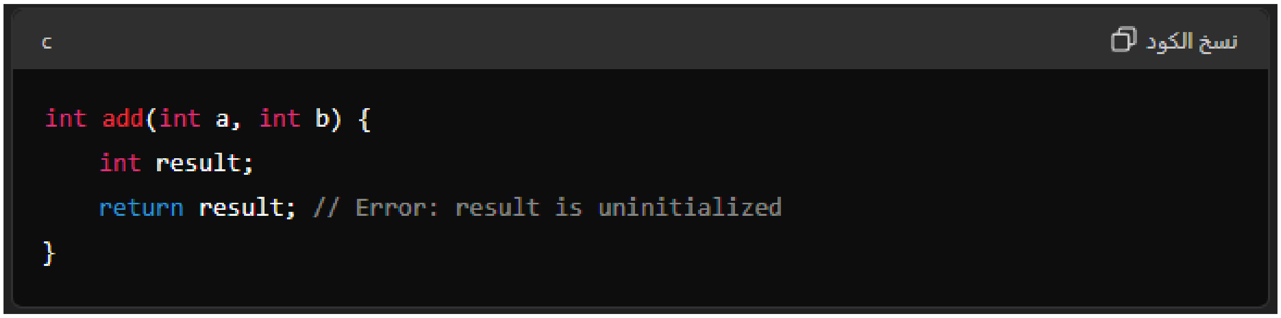
- **Error:** If the return type of a function does not match the expected type, it can lead to undefined behavior or incorrect results.
- **Solution:** Ensure that the return type matches the expected type.



```c
int factorial(int n) {
    return n * factorial(n - 1); // Error: No base case
}
```

Undefined Behavior with Uninitialized Variables:

- **Error:** Using uninitialized variables in a function can lead to undefined behavior.
- **Solution:** Ensure all variables are initialized before use.



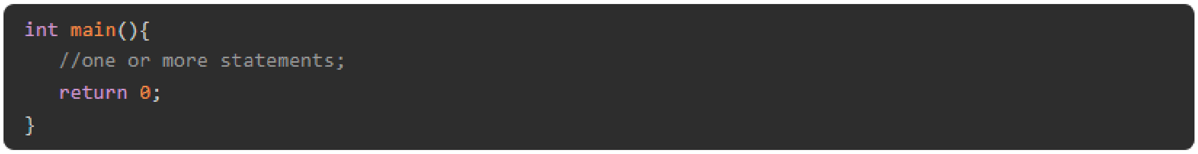
```c
int add(int a, int b) {
    int result;
    return result; // Error: result is uninitialized
}
```

main() Function in C

- ✓ The **main()** function in C is an entry point of any program.
- ✓ The program execution starts with the ***main()* function**.

In a C code, there may be any number of functions, but it must have a ***main()* function**. Irrespective of its place in the code, it is the first function to be executed.

Syntax

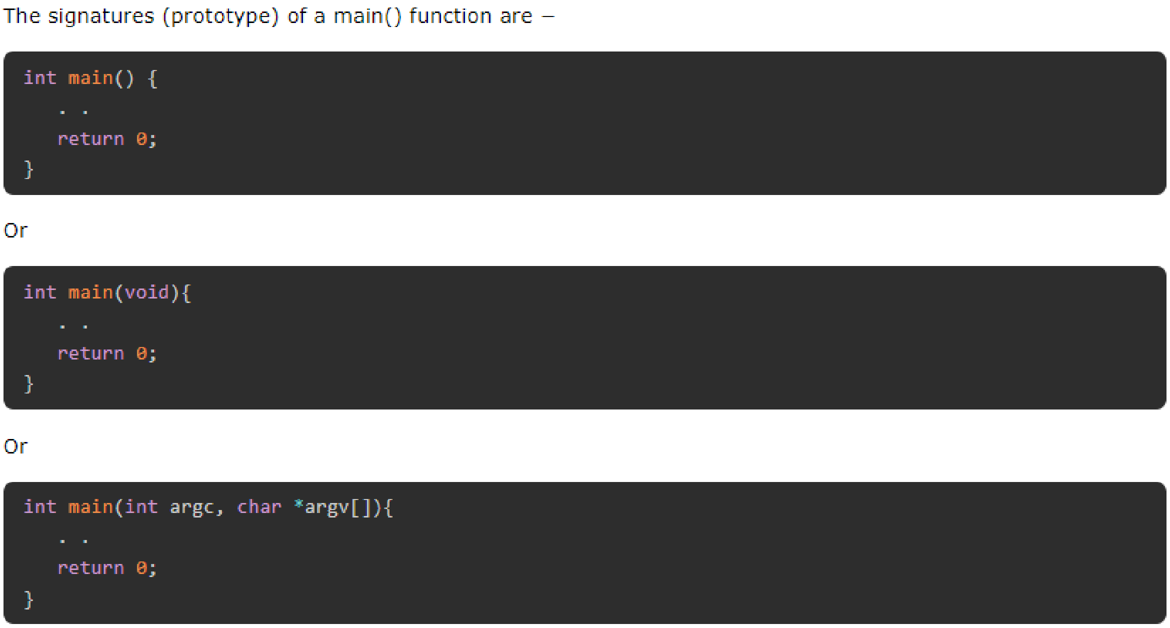


```
int main(){
   //one or more statements;
   return 0;
}
```

Syntax Explained

Typically, the main() function is defined with no arguments, although it may have arg and argv argument to receive values from the command line.

Valid/Different Signatures of main() Function



The signatures (prototype) of a main() function are –

```
int main() {
   . .
   return 0;
}
```

Or

```
int main(void){
   . .
   return 0;
}
```

Or

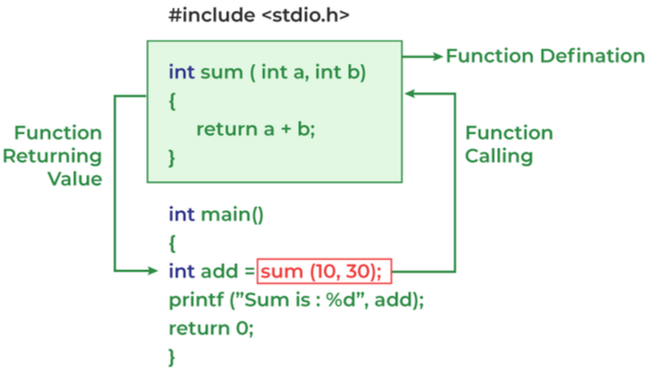
```
int main(int argc, char *argv[]){
   . .
   return 0;
}
```

Important Points about main() Function

1) A C program must have a main() function.
2) The main is not a C keyword.
3) It is classified as a user-defined function because its body is not pre-decided, it depends on the processing logic of the program.
4) By convention, int is the return type of main(). The last statement in the function body of main() returns 0, to indicate that the function has been successfully executed. Any non-zero return value indicates failure.
5) Some old C compilers let you define main() function with void return type However, this is considered to be non-standard and is not recommended.
6) As compared to other functions, the main() function:
   - ✓ Can't be declared as <u>inline</u>.
   - ✓ Can't be declared as <u>static</u>.
   - ✓ Can't have its address taken.
   - ✓ Can't be called from your program.

How does main() Works in C?

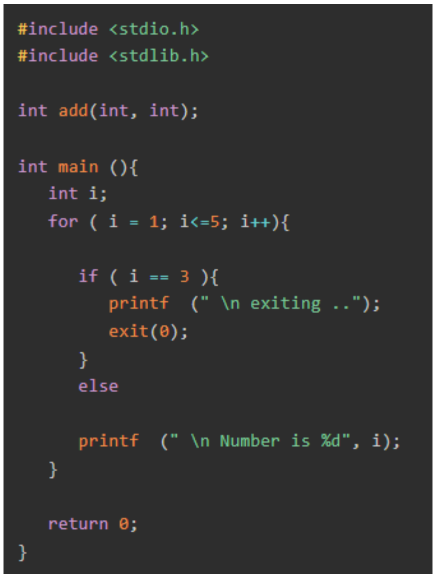
1) The main function can call any of these functions.
2) When main calls another function, it passes execution control to the function, optionally passing the requisite number and type of arguments, so that execution begins at the first statement in the called function.
3) The called function returns control to main when a return statement is executed or when the end of the function is reached.



```
#include <stdio.h>

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

int main()
{
int add = sum (10, 30);
printf ("Sum is : %d", add);
return 0;
}
```

Use of exit() in main() Function

- ✓ The C exit() function is a standard library function used to terminate the calling process.
- ✓ Use exit(0) to indicate no error, and exit(1) to indicate that the program is exiting with because of an error encountered.

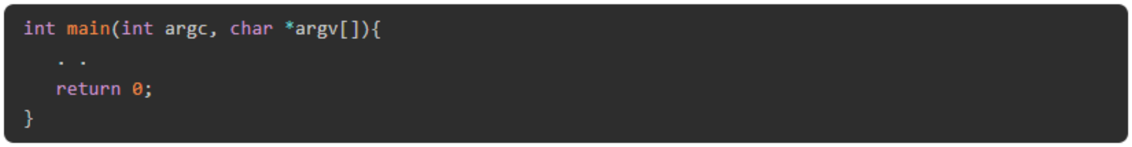


```
#include <stdio.h>
#include <stdlib.h>

int add(int, int);

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

      if ( i == 3 ){
         printf  (" \n exiting ..");
         exit(0);
      }
      else

      printf  (" \n Number is %d", i);
   }

   return 0;
}
```

Command-line Arguments with main()

- ✓ Typically, the **main()** function is defined without any arguments.
- ✓ you may define main() with arguments to let it accept the values from the command line

Syntax



```
int main(int argc, char *argv[]){
   . .
   return 0;
}
```

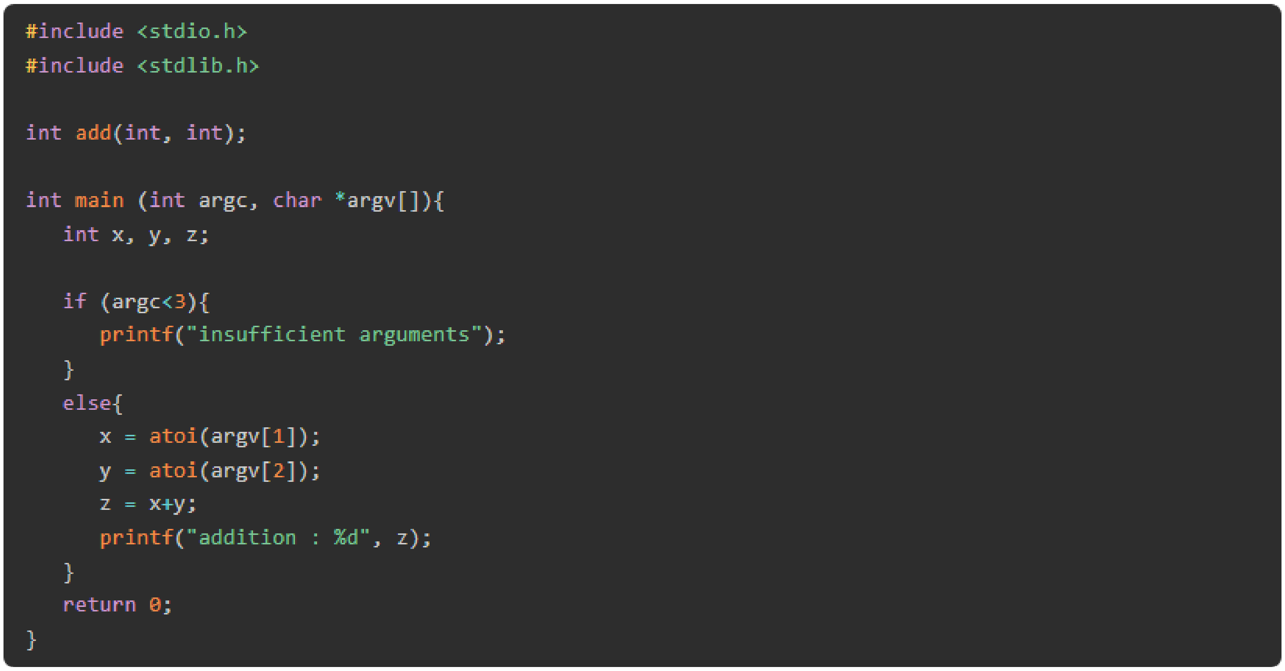
Argument Definitions

argc – The first argument is an integer that contains the count of arguments that follow in argv. The argc parameter is always greater than or equal to 1.

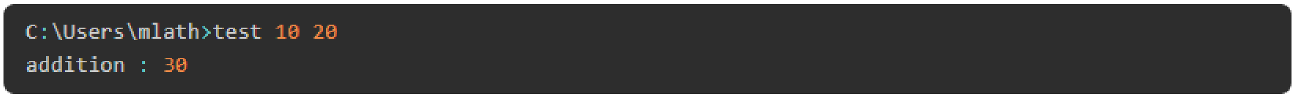
argv – The second argument is an array of null–terminated strings representing command-line arguments entered by the user of the program.

- ✓ By convention, argv[0] is the command with which the program is invoked.
- ✓ argv[1] is the first command–line argument.
- ✓ The last argument from the command line is argv[argc – 1], and argv[argc] is always NULL.

Example: Using main() Function with Command-line Arguments



```
#include <stdio.h>
#include <stdlib.h>

int add(int, int);

int main (int argc, char *argv[]){
   int x, y, z;

   if (argc<3){
      printf("insufficient arguments");
   }
   else{
      x = atoi(argv[1]);
      y = atoi(argv[2]);
      z = x+y;
      printf("addition : %d", z);
   }
   return 0;
}
```



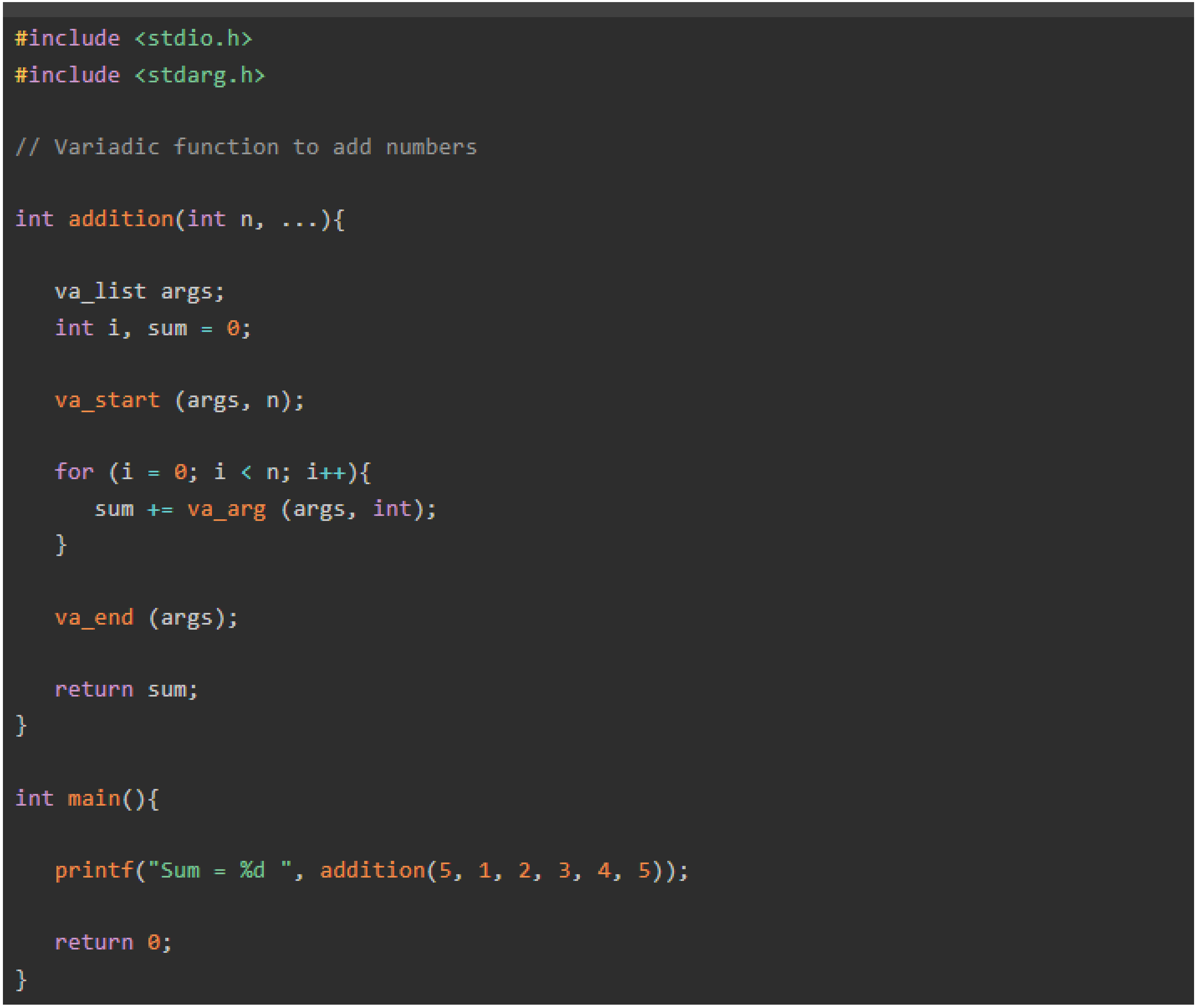
```
C:\Users\mlath>test 10 20
addition : 30
```

Variadic Functions in C

A function that can take a variable number of arguments is called a variadic function. One fixed argument is required to define a variadic function.

Methods	Description
va_start(va_list ap, arg)	Arguments after the last fixed argument are stored in the va_list.
va_arg(va_list ap, type)	Each time, the next argument in the variable list va_list and coverts it to the given type, till it reaches the end of list.
va_copy(va_list dest, va_list src)	This creates a copy of the arguments in va_list
va_end(va_list ap)	This ends the traversal of the variadic function arguments. As the end of va_list is reached, the list object is cleaned up.

Example: Sum of Numbers Using a Variadic Function

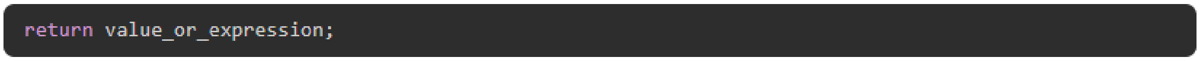


```c
#include <stdio.h>
#include <stdarg.h>

// Variadic function to add numbers

int addition(int n, ...){

   va_list args;
   int i, sum = 0;

   va_start (args, n);

   for (i = 0; i < n; i++){
      sum += va_arg (args, int);
   }

   va_end (args);

   return sum;
}

int main(){

   printf("Sum = %d ", addition(5, 1, 2, 3, 4, 5));

   return 0;
}
```

Return Statement in C

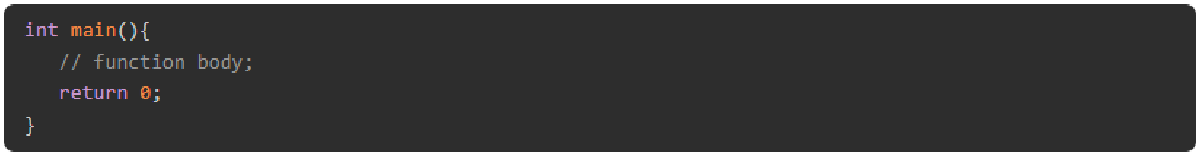
- ✓ Every function should have a **return** statement as its last statement.
- ✓ While using the returns statement, the return type and returned value (expression) must be the same.

Syntax of return Statement



```
return value_or_expression;
```

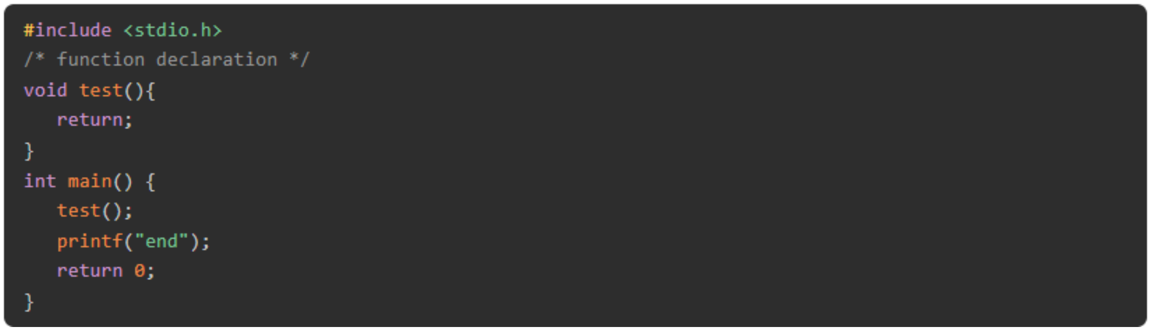
The following **main()** function shows return as its last statement –



```
int main(){
   // function body;
   return 0;
}
```

The void return statement

A function's return type can be **void**. In such a case, return statement is optional. It may be omitted, or return without any expression is used.

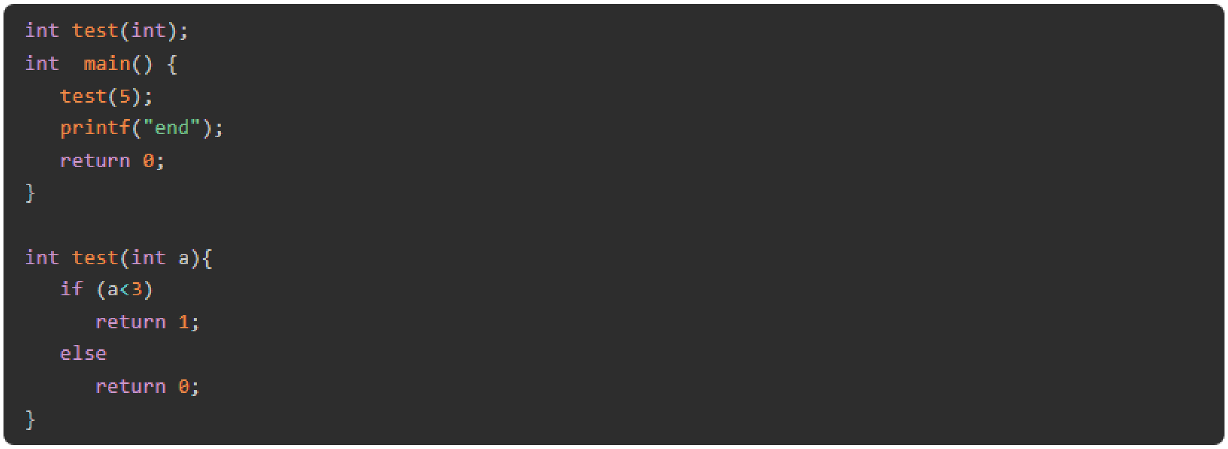


```
#include <stdio.h>
/* function declaration */
void test(){
   return;
}
int main() {
   test();
   printf("end");
   return 0;
}
```

Multiple return values with return statement

A function can be defined with more than one arguments, but can return only one value. You can however use multiple conditional return statements as shown below –

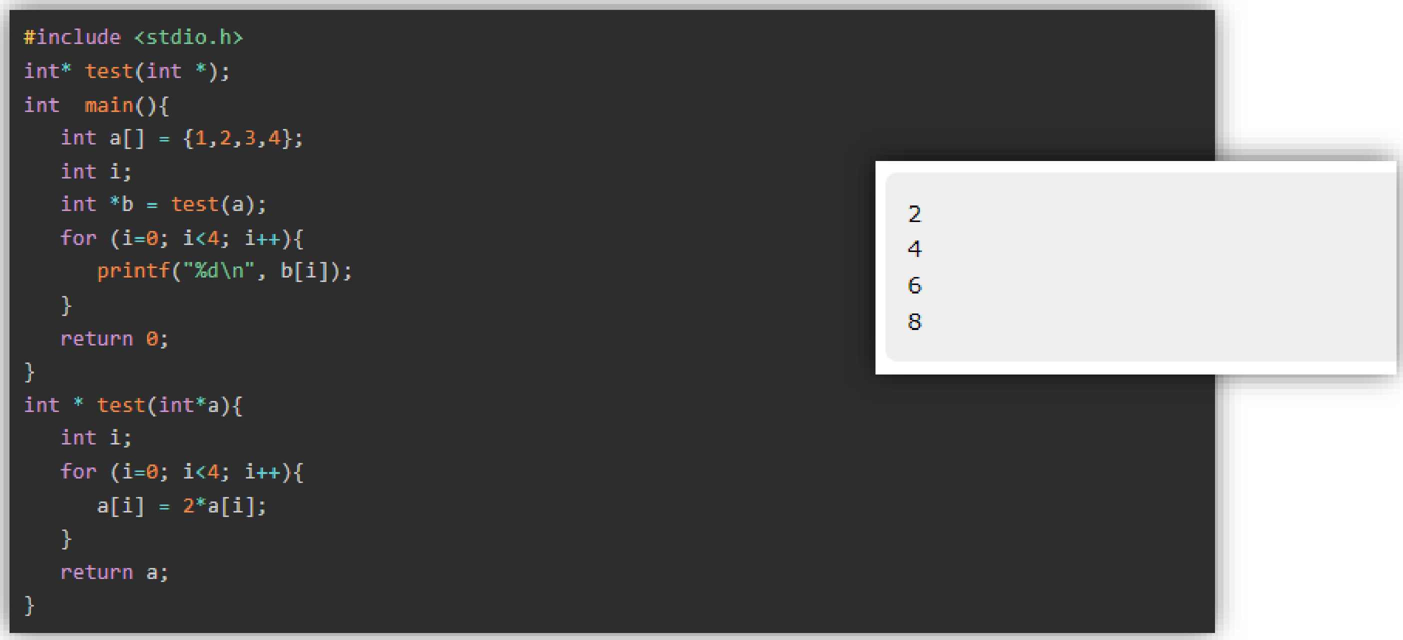
Example



```
int test(int);
int  main() {
   test(5);
   printf("end");
   return 0;
}

int test(int a){
   if (a<3)
      return 1;
   else
      return 0;
}
```

Function returning an array

It is not possible to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.



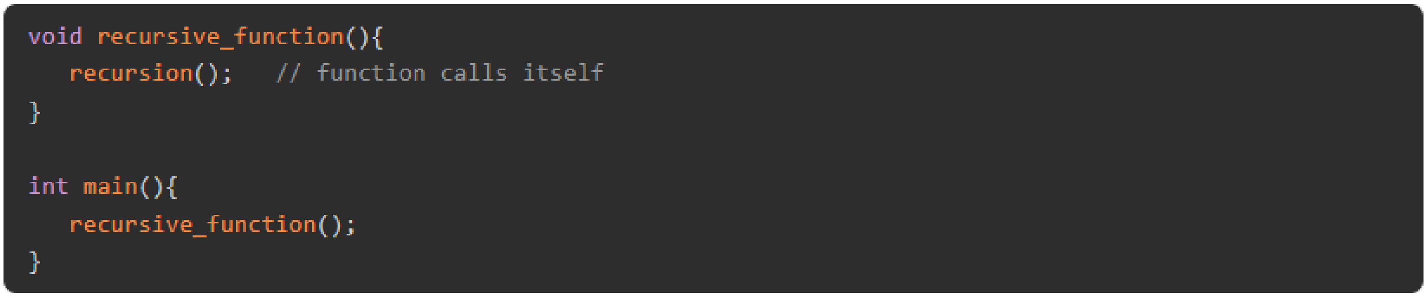
```
#include <stdio.h>
int* test(int *);
int  main(){
   int a[] = {1,2,3,4};
   int i;
   int *b = test(a);
   for (i=0; i<4; i++){
      printf("%d\n", b[i]);
   }
   return 0;
}
int * test(int*a){
   int i;
   for (i=0; i<4; i++){
      a[i] = 2*a[i];
   }
   return a;
}
```

```
2
4
6
8
```

Recursion in C

- ✓ **Recursion** is the process by which a function calls itself.
- ✓ C language allows writing of such functions which call itself to solve complicated problems by breaking them down into simple and easy problems.
- ✓ These functions are known as **recursive functions**.

Syntax



```
void recursive_function(){
   recursion();    // function calls itself
}

int main(){
   recursive_function();
}
```

Passing Array to Function in C

First way:

```
return_type function(type arrayname[])
```

Second way:

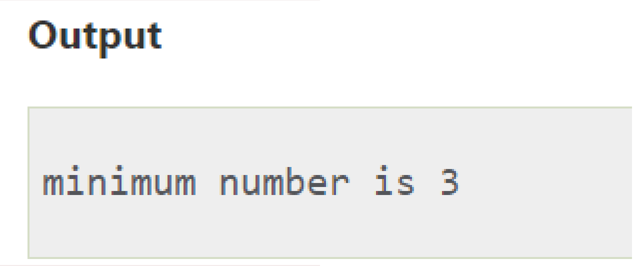
```
return_type function(type arrayname[SIZE])
```

Third way:

```
return_type function(type *arrayname)
```

example

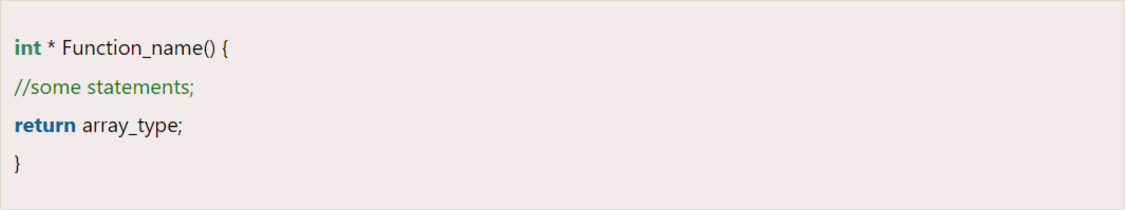
```
#include<stdio.h>
int minarray(int arr[],int size){
int min=arr[0];
int i=0;
for(i=1;i<size;i++){
if(min>arr[i]){
min=arr[i];
}
}//end of for
return min;
}//end of function

int main(){
int i=0,min=0;
int numbers[]={4,5,7,3,8,9};//declaration of array

min=minarray(numbers,6);//passing array with size
printf("minimum number is %d \n",min);
return 0;
}
```



Output

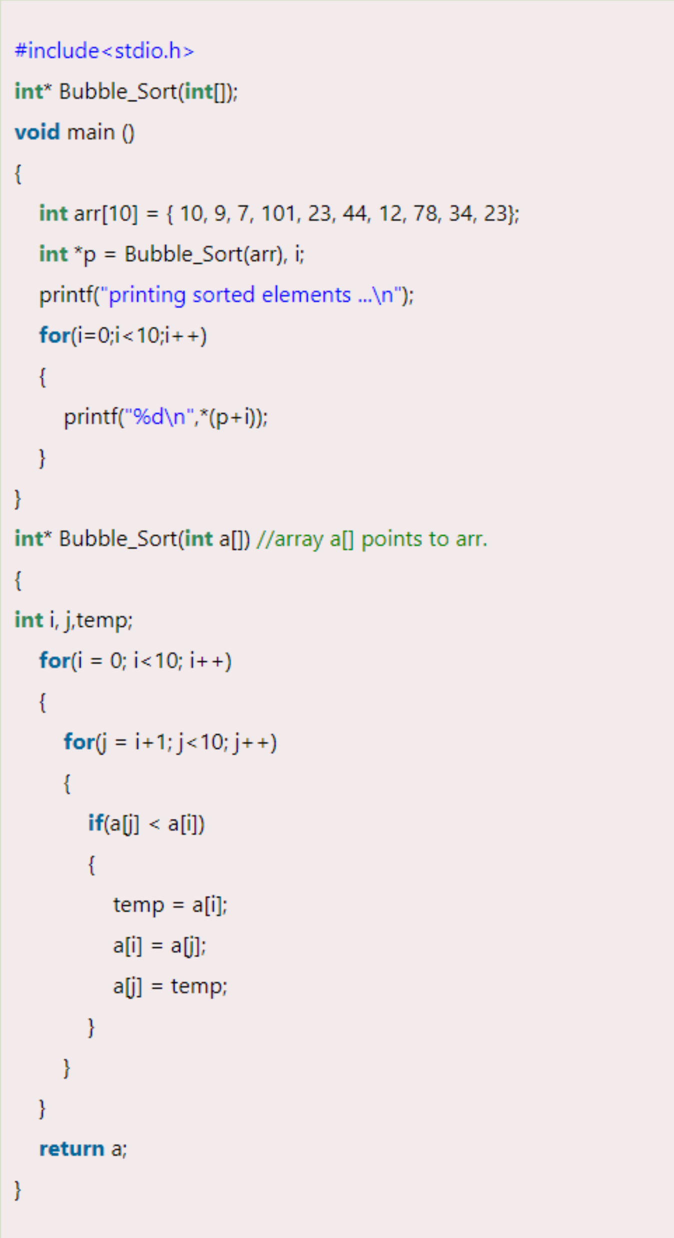
```
minimum number is 3
```

Returning array from the function

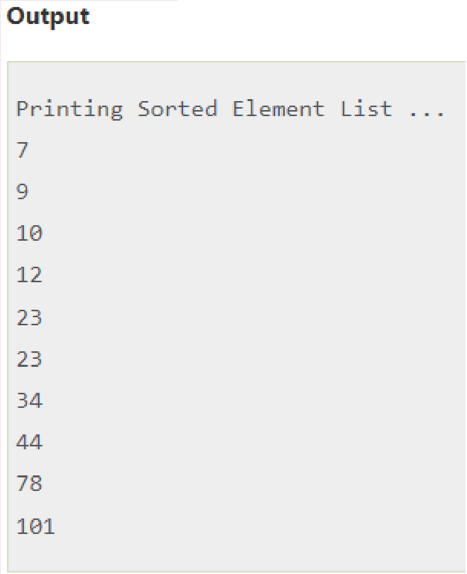


```
int * Function_name() {
//some statements;
return array_type;
}
```

example



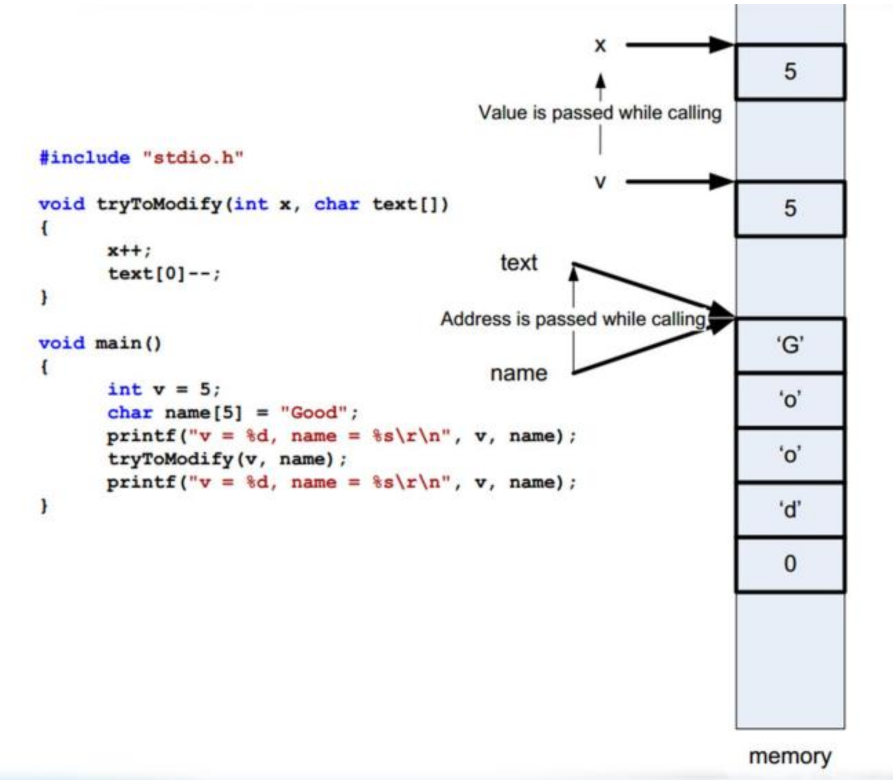
```
#include<stdio.h>
int* Bubble_Sort(int[]);
void main ()
{
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int *p = Bubble_Sort(arr), i;
    printf("printing sorted elements ...\n");
    for(i=0;i<10;i++)
    {
        printf("%d\n",*(p+i));
    }
}
int* Bubble_Sort(int a[]) //array a[] points to arr.
{
int i, j,temp;
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    return a;
}
```



Output

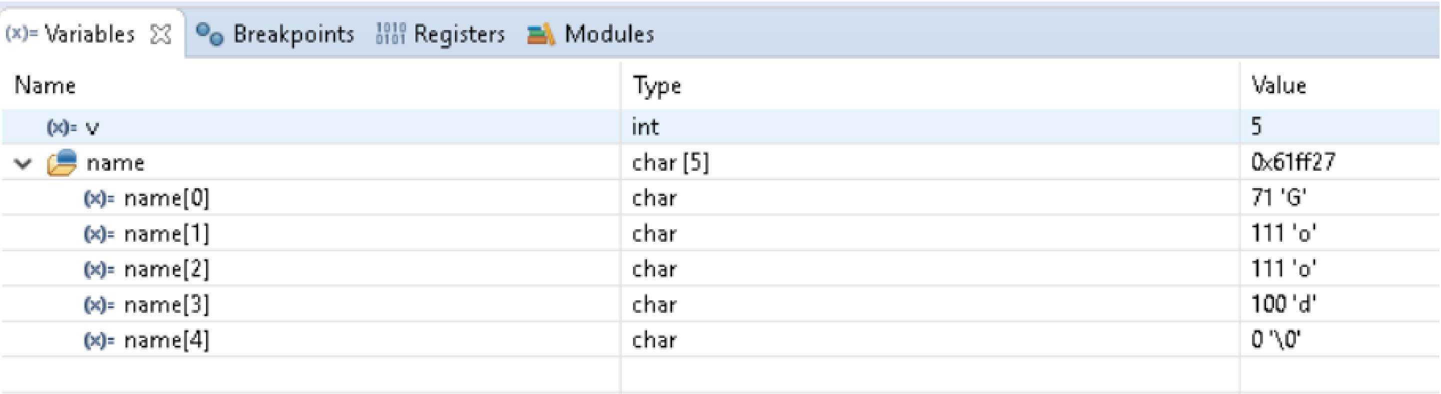
```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

example

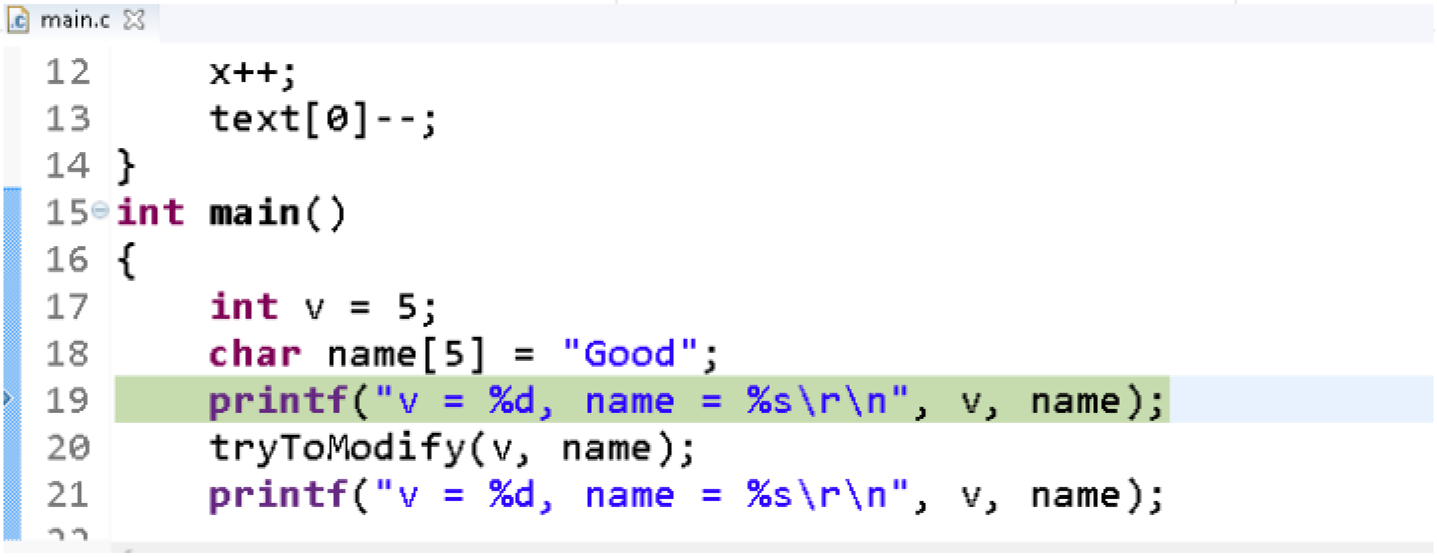


```c
#include "stdio.h"

void tryToModify(int x, char text[])
{
        x++;
        text[0]--;
}

void main()
{
        int v = 5;
        char name[5] = "Good";
        printf("v = %d, name = %s\r\n", v, name);
        tryToModify(v, name);
        printf("v = %d, name = %s\r\n", v, name);
}
```

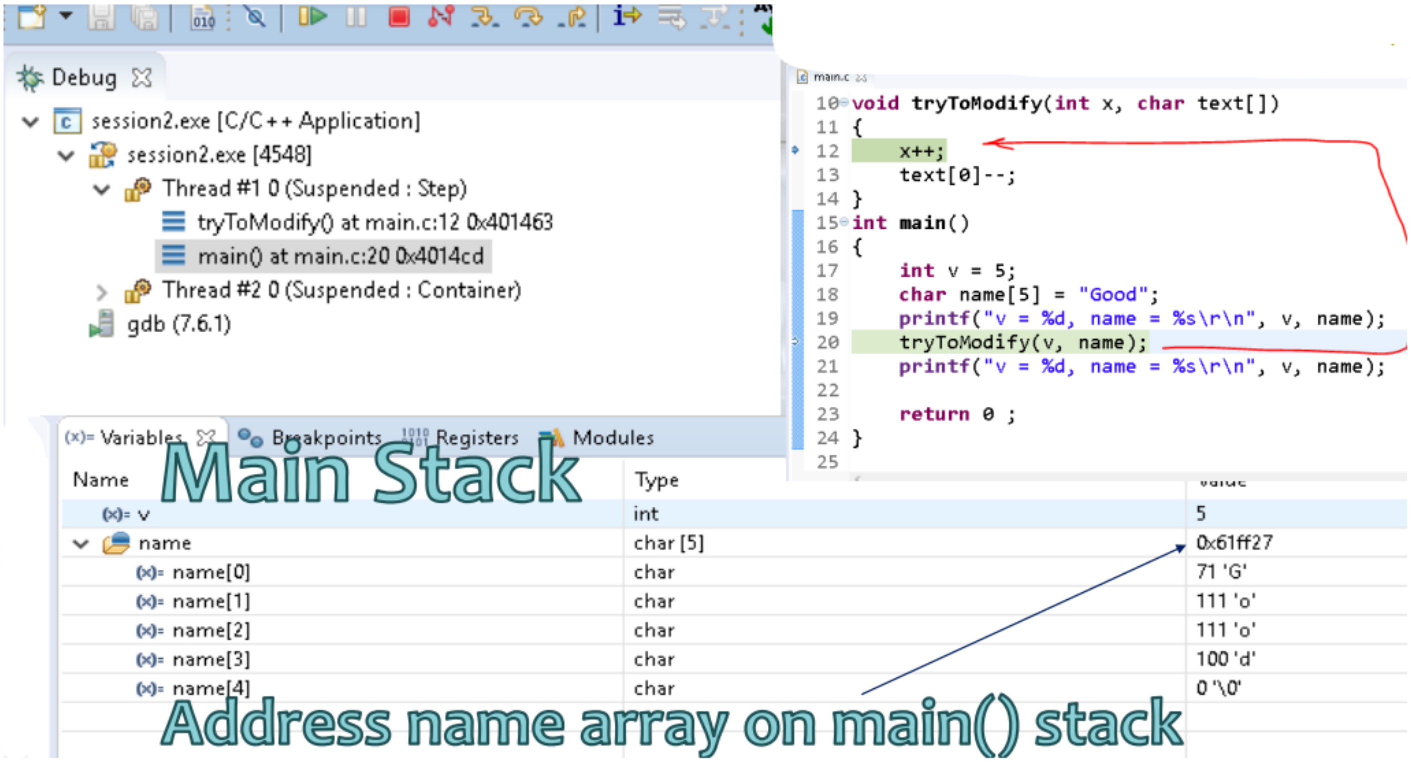
x
5
Value is passed while calling
v
5
text
Address is passed while calling
name
'G'
'o'
'o'
'd'
0
memory



Name	Type	Value
v	int	5
name	char [5]	0x61ff27
name[0]	char	71 'G'
name[1]	char	111 'o'
name[2]	char	111 'o'
name[3]	char	100 'd'
name[4]	char	0 '\0'



```c
    x++;
    text[0]--;
}
int main()
{
    int v = 5;
    char name[5] = "Good";
    printf("v = %d, name = %s\r\n", v, name);
    tryToModify(v, name);
    printf("v = %d, name = %s\r\n", v, name);
```



Debug

- session2.exe [C/C++ Application]
  - session2.exe [4548]
    - Thread #1 0 (Suspended : Step)
      - tryToModify() at main.c:12 0x401463
      - main() at main.c:20 0x4014cd
    - Thread #2 0 (Suspended : Container)
  - gdb (7.6.1)

main.c

```
10 void tryToModify(int x, char text[])
11 {
12     x++;
13     text[0]--;
14 }
15 int main()
16 {
17     int v = 5;
18     char name[5] = "Good";
19     printf("v = %d, name = %s\r\n", v, name);
20     tryToModify(v, name);
21     printf("v = %d, name = %s\r\n", v, name);
22 
23     return 0 ;
24 }
25 
```

Name	Type	Value
v	int	5
name	char [5]	0x61ff27
name[0]	char	71 'G'
name[1]	char	111 'o'
name[2]	char	111 'o'
name[3]	char	100 'd'
name[4]	char	0 '\0'

What is the purpose of main() function?

- ✓ In C, program execution starts from the main() function.
- ✓ Every C program must contain a main() function. The main function may contain any number of statements.
- ✓ These statements are executed sequentially in the order which they are written.
- ✓ The main function can in-turn call other functions. When main calls a function, it passes the execution control to that function.
- ✓ The function returns control to main when a return statement is executed or when end of function is reached.

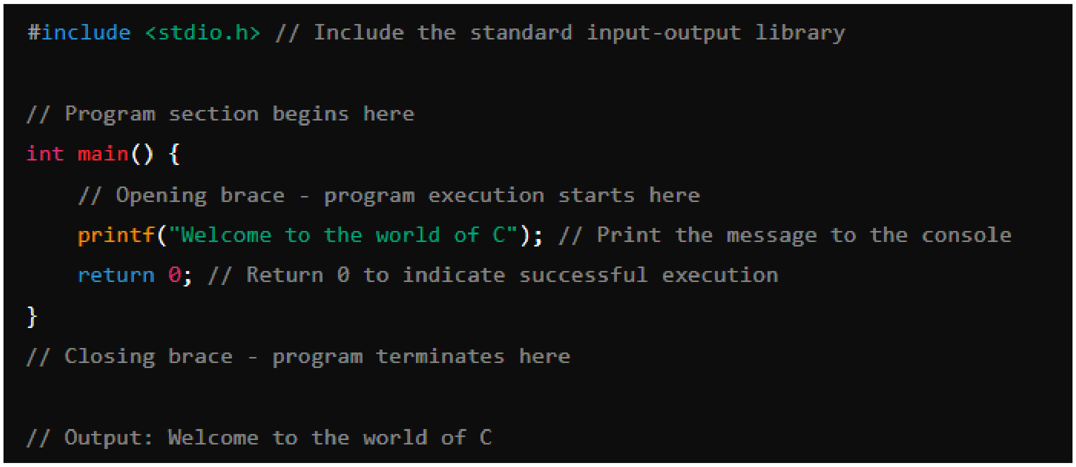
In C, the function prototype of the 'main' is one of the following:

int main(); //main with no arguments

int main(int argc, char *argv[]); //main with arguments

The parameters argc and argv respectively give the number and value of the program's command-line arguments.

Example:



```
#include <stdio.h> // Include the standard input-output library

// Program section begins here
int main() {
    // Opening brace - program execution starts here
    printf("Welcome to the world of C"); // Print the message to the console
    return 0; // Return 0 to indicate successful execution
}
// Closing brace - program terminates here

// Output: Welcome to the world of C
```

Explain command line arguments of main function?

- ✓ In C, we can supply arguments to 'main' function.
- ✓ The arguments that we pass to main ( ) at command prompt are called command line arguments.
- ✓ These arguments are supplied at the time of invoking the program

- ❖ The main ( ) function can take arguments as: main(int argc, char *argv[]) { }

The first argument argc is known as 'argument counter'.

It represents the number of arguments in the command line. The second argument argv is known as 'argument vector'. It is an array of char type pointers that points to the command line arguments. Size of this array will be equal to the value of argc.

Example: at the command prompt if we give

C:\> fruit.exe apple mango

Then

argc would contain value 3

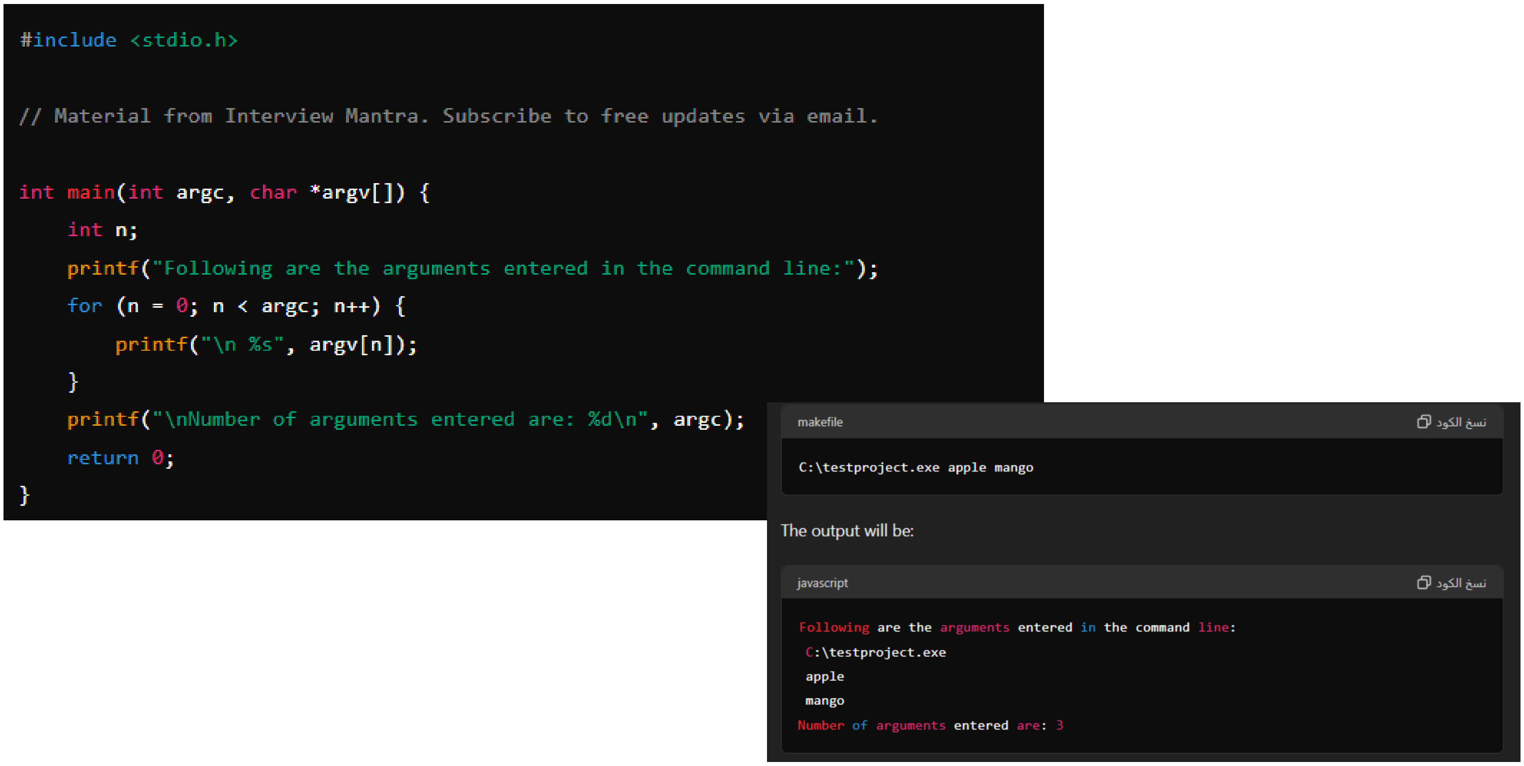
argv [0] would contain base address of string " fruit.exe" which is the command name that invokes the program.

argv [1] would contain base address of string "apple"

argv [2] would contain base address of string "mango"

here apple and mango are the arguments passed to the program fruit.exe

Program:



```c
#include <stdio.h>

// Material from Interview Mantra. Subscribe to free updates via email.

int main(int argc, char *argv[]) {
    int n;
    printf("Following are the arguments entered in the command line:");
    for (n = 0; n < argc; n++) {
        printf("\n %s", argv[n]);
    }
    printf("\nNumber of arguments entered are: %d\n", argc);
    return 0;
}
```

```
C:\testproject.exe apple mango
```

The output will be:

```
Following are the arguments entered in the command line:
 C:\testproject.exe
 apple
 mango
Number of arguments entered are: 3
```

What are header files? Are functions declared or defined in header files ?

Functions and macros are declared in header files. Header files would be included in source files by the compiler at the time of compilation.

Header files are included in source code using #include directive.#include includes all the declarations present in the header file 'some.h'.

A header file may contain declarations of sub-routines, functions, macros and also variables which we may want to use in our program. Header files help in reduction of repetitive code.

Syntax of include directive:

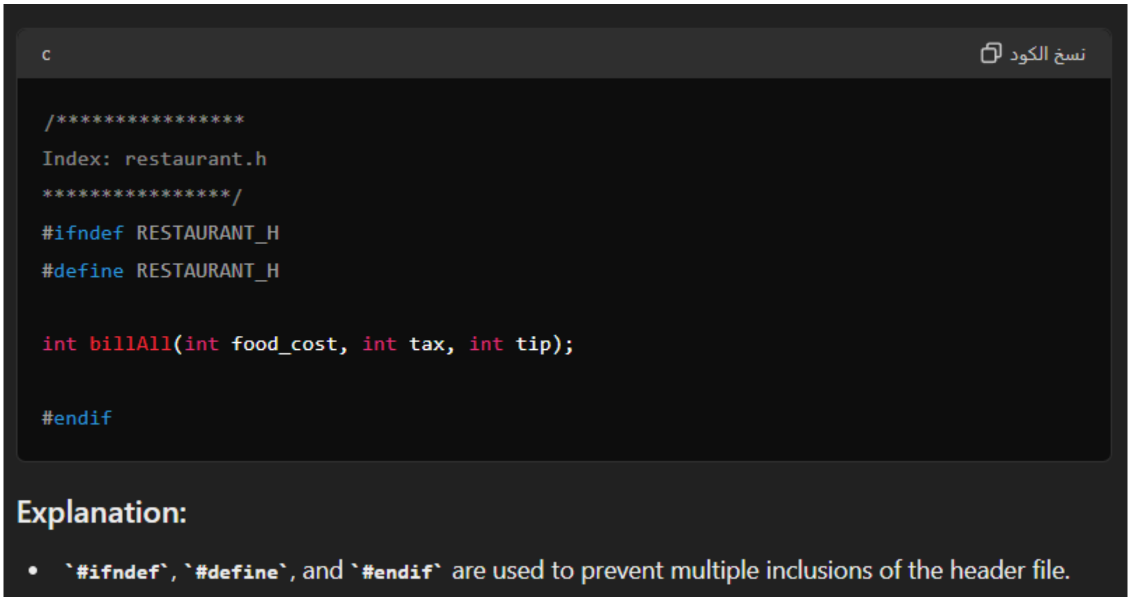
#include <stdio.h>//includes the header file stdio.h, standard input output header into the source code

Functions can be declared as well as defined in header files. But it is recommended only to declare functions and not to define in the header files. When we include a header file in our program we actually are including all the functions, macros and variables declared in it

In case of pre-defined C standard library header files ex(stdio.h), the functions calls are replaced by equivalent binary code present in the pre-compiled libraries. Code for C standard functions are linked and then the program is executed. Header files with custom names can also be created.

Custom Header Files Example

- restaurant.h
- This header file declares the function billAll.

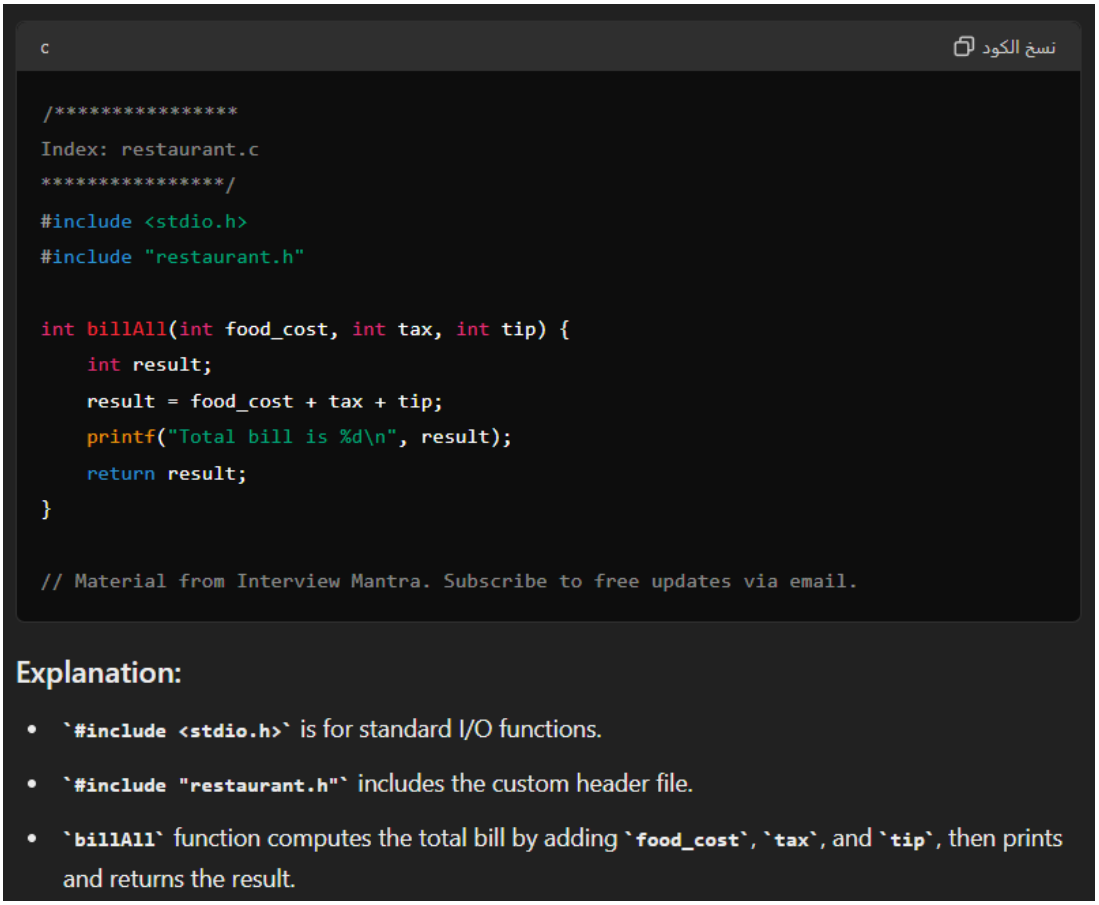


```c
/****************
Index: restaurant.h
****************/
#ifndef RESTAURANT_H
#define RESTAURANT_H

int billAll(int food_cost, int tax, int tip);

#endif
```

Explanation:

- `#ifndef`, `#define`, and `#endif` are used to prevent multiple inclusions of the header file.

- *restaurant.c*
- This source file defines the function billAll.

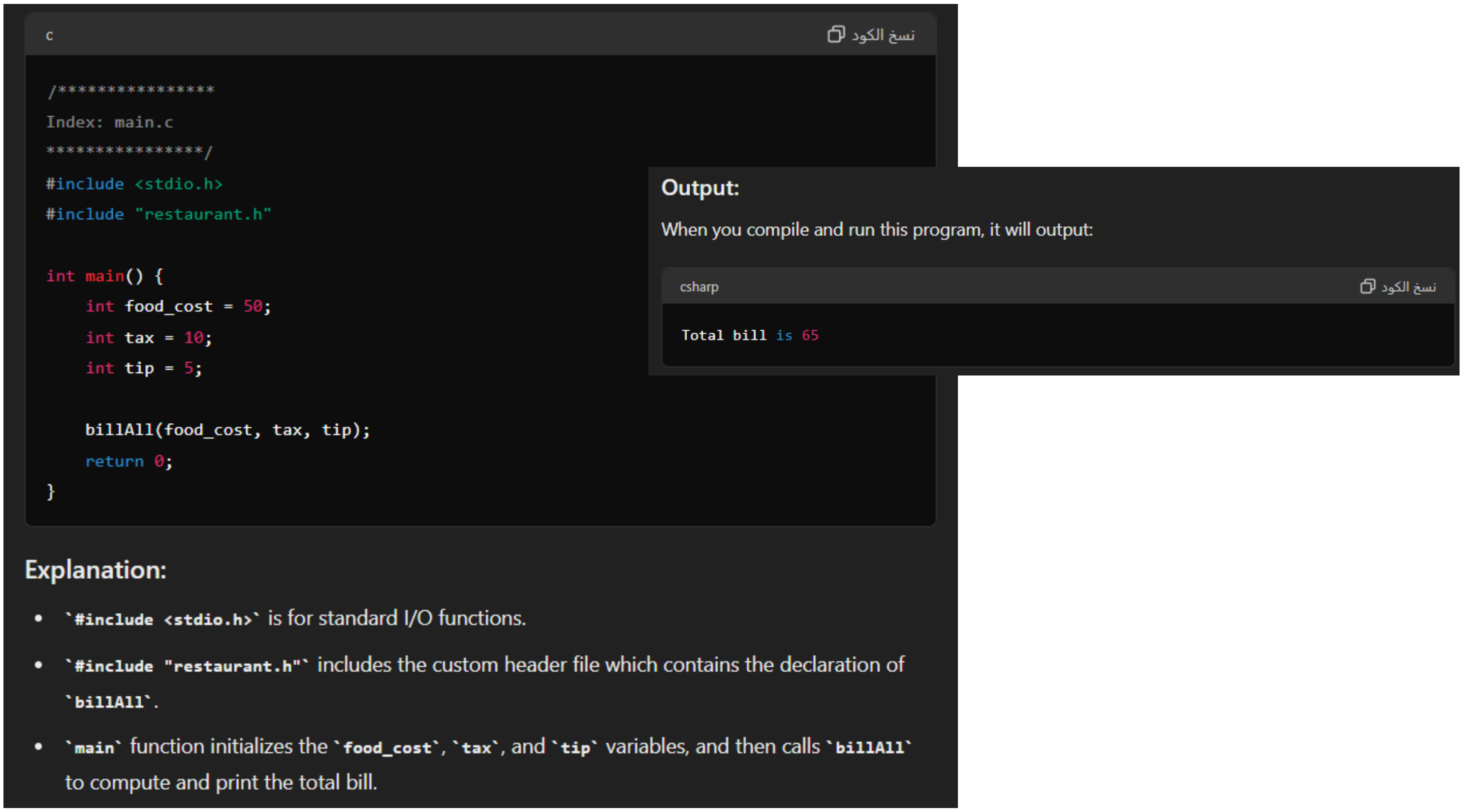


```c
/****************
Index: restaurant.c
****************/
#include <stdio.h>
#include "restaurant.h"

int billAll(int food_cost, int tax, int tip) {
    int result;
    result = food_cost + tax + tip;
    printf("Total bill is %d\n", result);
    return result;
}

// Material from Interview Mantra. Subscribe to free updates via email.
```

Explanation:

- `#include <stdio.h>` is for standard I/O functions.
- `#include "restaurant.h"` includes the custom header file.
- `billAll` function computes the total bill by adding `food_cost`, `tax`, and `tip`, then prints and returns the result.

- *main.c*
- This source file uses the function declared in restaurant.h and defined in restaurant.c.



```c
/****************
Index: main.c
****************/
#include <stdio.h>
#include "restaurant.h"

int main() {
    int food_cost = 50;
    int tax = 10;
    int tip = 5;

    billAll(food_cost, tax, tip);
    return 0;
}
```

Output:

When you compile and run this program, it will output:

```
Total bill is 65
```

Explanation:

- `#include <stdio.h>` is for standard I/O functions.
- `#include "restaurant.h"` includes the custom header file which contains the declaration of `billAll`.
- `main` function initializes the `food_cost`, `tax`, and `tip` variables, and then calls `billAll` to compute and print the total bill.

What are the differences between formal arguments and actual arguments of a function?

Argument: An argument is an expression which is passed to a function by its caller (or macro by its invoker) in order for the function(or macro) to perform its task. It is an expression in the comma-separated list bound by the parentheses in a function call expression

Actual arguments: The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Formal arguments: The formal arguments are the parameters/arguments in a function declaration. The scope of formal arguments is local to the function definition in which they are used. Formal arguments belong to the called function. Formal arguments are a copy of the actual arguments. A change in formal arguments would not be reflected in the actual arguments.

What is pass by value in functions?

Pass by Value: In this method, the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. In pass by value, the changes made to formal arguments in the called function have no effect on the values of actual arguments in the calling function.

What is pass by reference in functions?

Pass by Reference: In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses, we would have an access to the actual arguments and hence we would be able to manipulate them. C does not support Call by reference. But it can be simulated using pointers.

Abdallah Ghazy

Abdallah-Ghazy

Mastering Embedded Systems for Innovation and Success

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"