C - FUNCTION

Dr. Sheak Rashed Haider Noori

Professor & Associate Head

Department of Computer Science

OUTLINE

- What is C function?
- Uses of C functions
- C function declaration, function call and definition with example program
- How to call C functions in a program?
  - Call by value
  - Call by reference
- C function arguments and return values
  - C function with arguments and with return value
  - C function with arguments and without return value
  - C function without arguments and without return value
  - C function without arguments and with return value
- Types of C functions
  - Library functions in C
  - User defined functions in C
    - Creating/Adding user defined function in C library
- Command line arguments in C
- Variable length arguments in C

WHAT IS C FUNCTION?

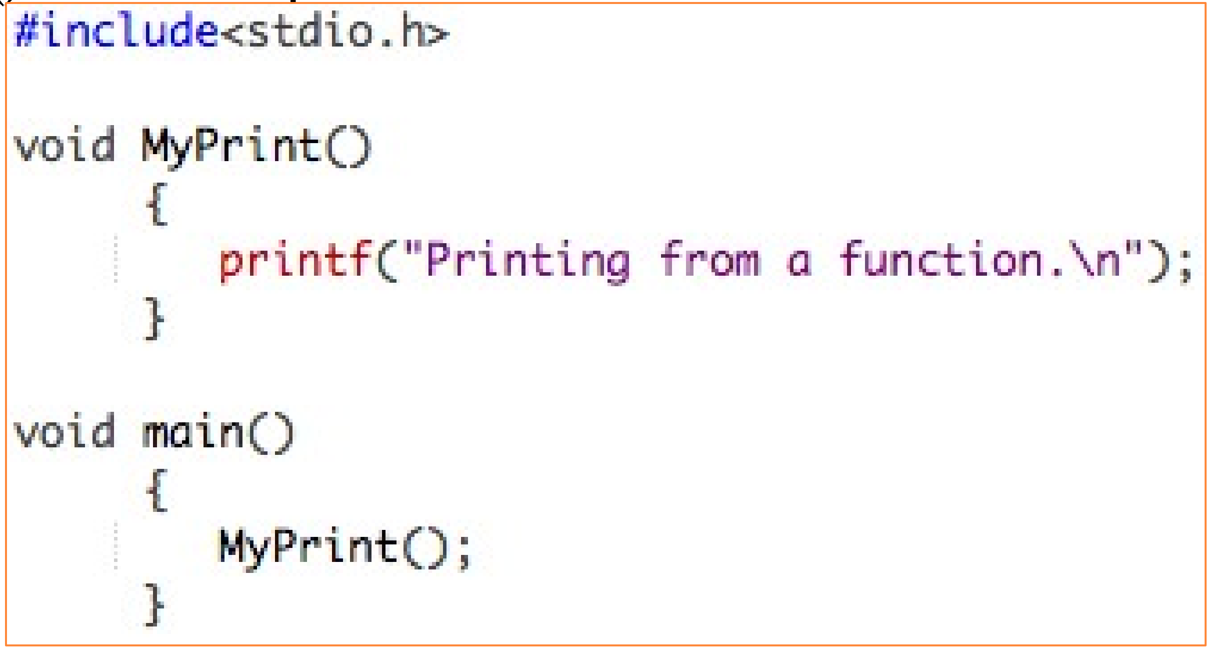
- A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.
- 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.
- The C standard library provides numerous built-in functions that your program can call. For example, function **printf()** to print output in the console.
- A function is known with various names like a *method* or a *sub-routine* or a *procedure*, etc.

C - FUNCTIONS

- Most languages allow you to create functions of some sort.
- Functions are used to break up large programs into named sections.
- You have already been using a function which is the *main* function.
- Functions are often used when the same piece of code has to run multiple times.
- In this case you can put this piece of code in a function and give that function a name. When the piece of code is required you just have to call the function by its name. (So you only have to type the piece of code once).

C - FUNCTIONS

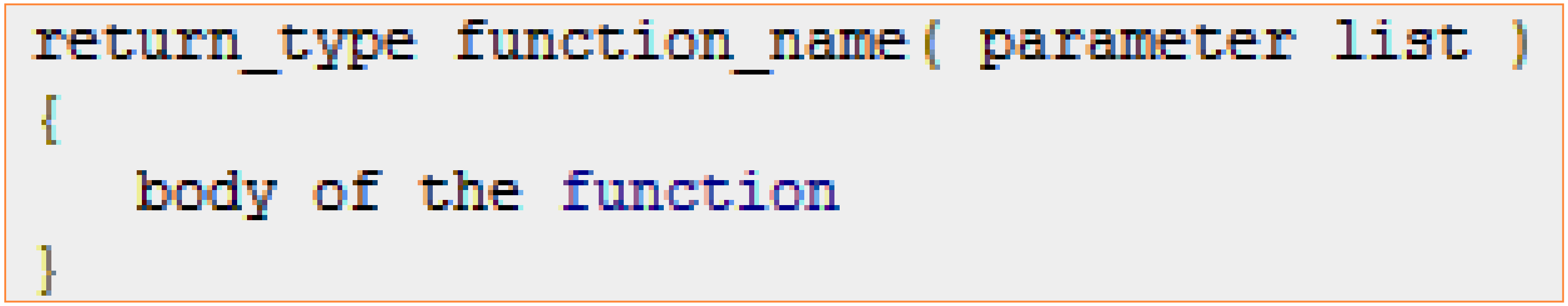
- In the example below we declare a function with the name MyPrint.
- The only thing that this function does is to print the sentence: “Printing from a function”.
- If we want to use the function we just have to call MyPrint() and the printf statement will be executed.



```
#include<stdio.h>

void MyPrint()
     {
          printf("Printing from a function.\n");
     }

void main()
     {
          MyPrint();
     }
```

DEFINING A FUNCTION

- The general form of a function definition is as follows:



```
return_type function_name( parameter list )
{
   body of the function
}
```

- A function definition in C language consists of
  - a *function header* and
  - a *function body*.

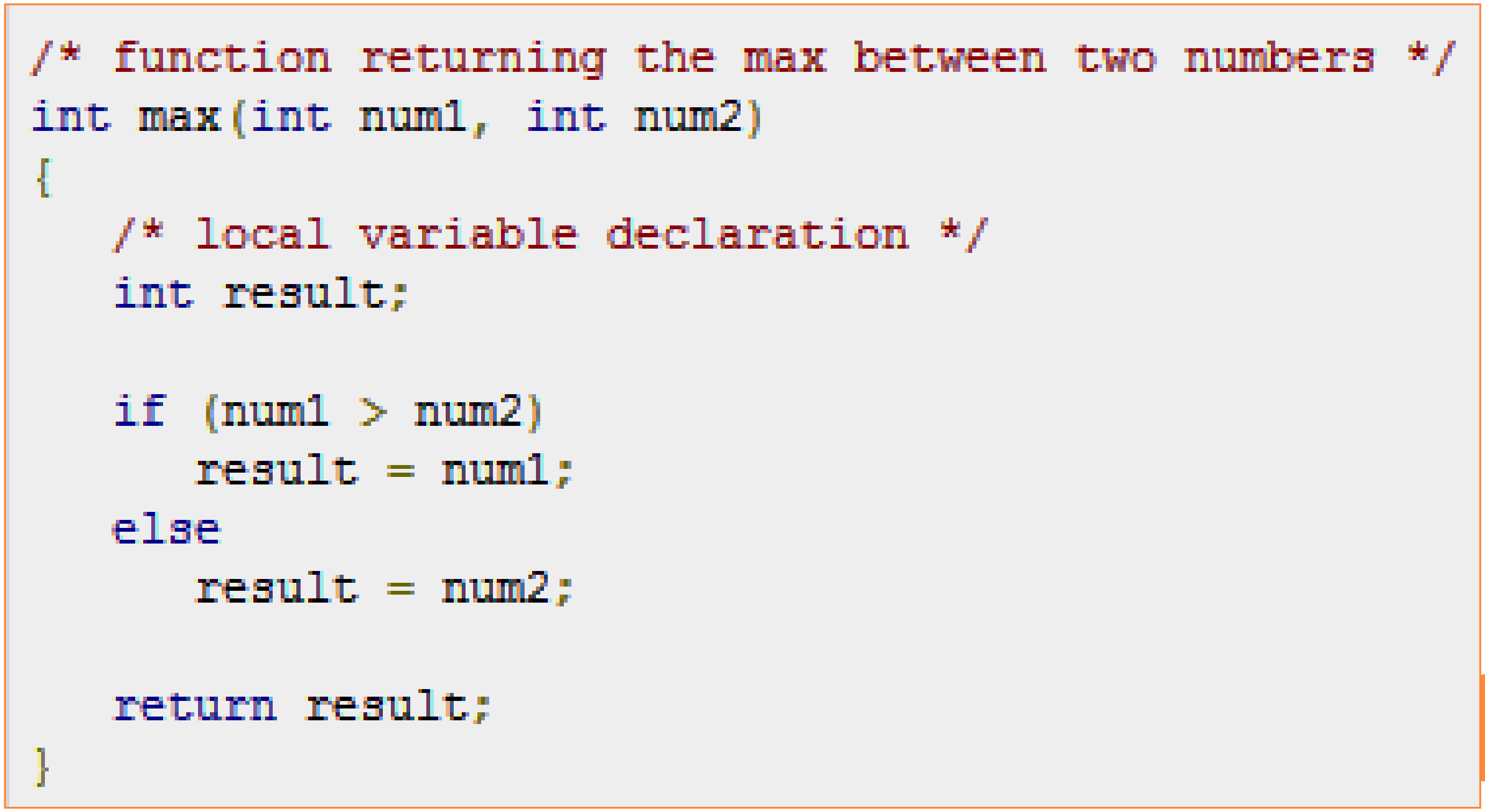
PARTS OF A FUNCTION

Here are all the parts of a function:

- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as ***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.
- **Return Type**: A function may return a value. The **return type** is the data type of the value the function returns. If you don't want to return a result from a function, you can use void return type. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword **void**.
- **Function Body:** The function body contains a collection of statements that define what the function does.

EXAMPLE: FUNCTION

- Following is the source code for a function called **max()**. This function takes two parameters **num1** and **num2** and returns the maximum between the two:



```c
/* 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;
}
```

FUNCTION DECLARATIONS

- A function **declaration(function prototype)** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts:

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

- For the above defined function *max*(), following is the function declaration:

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

- Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

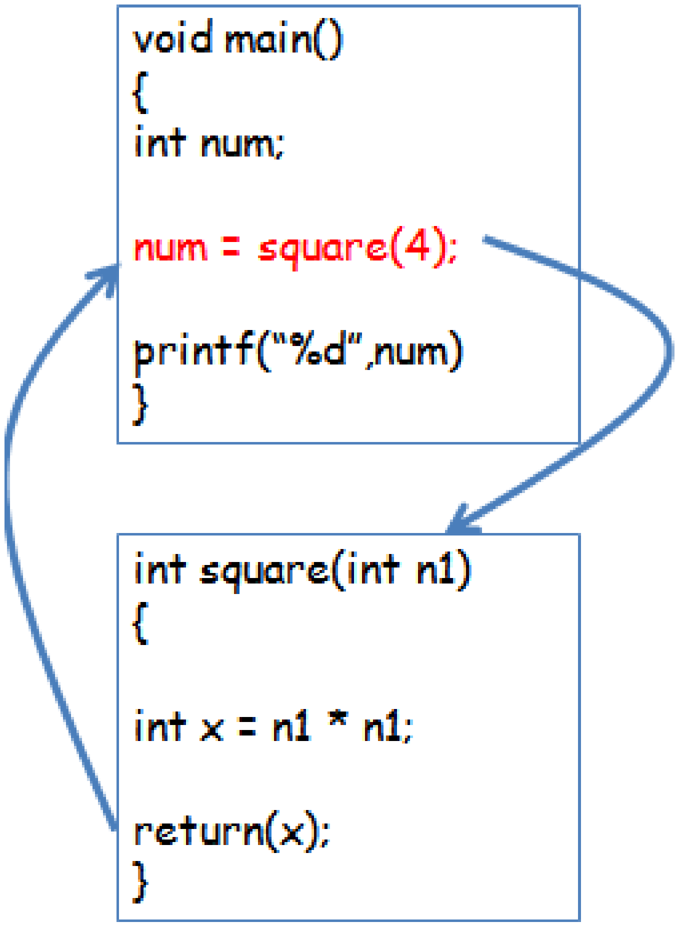
```
int max(int, int);
```

- Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

CALLING A FUNCTION

- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its *return* statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

CALLING A FUNCTION



```
void main()
{
int num;

num = square(4);

printf("%d",num)
}
```

```
int square(int n1)
{

int x = n1 * n1;

return(x);
}
```

Understanding Flow

1. Firstly Operating System will call our main function.
2. When control comes inside main function , execution of main starts (i.e execution of C program starts)
3. Consider Line 4 :

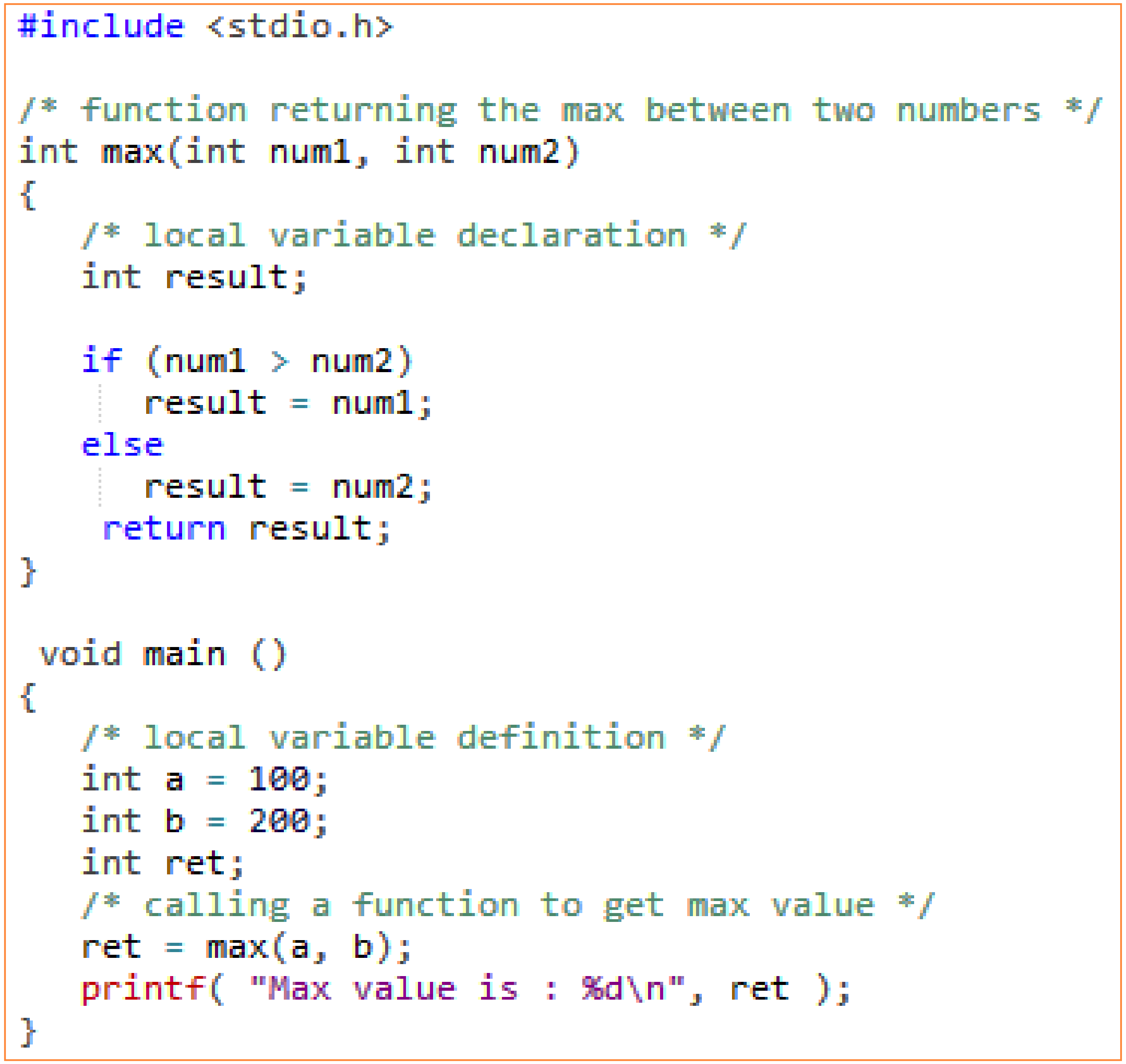
```
num = square(4);
```

4. We have called a function square(4). [See : How to call a function ?].
5. We have passed "4" as parameter to function.

Note : Calling a function halts execution of the current function , it will execute called function , after execution control returned back to the calling function.

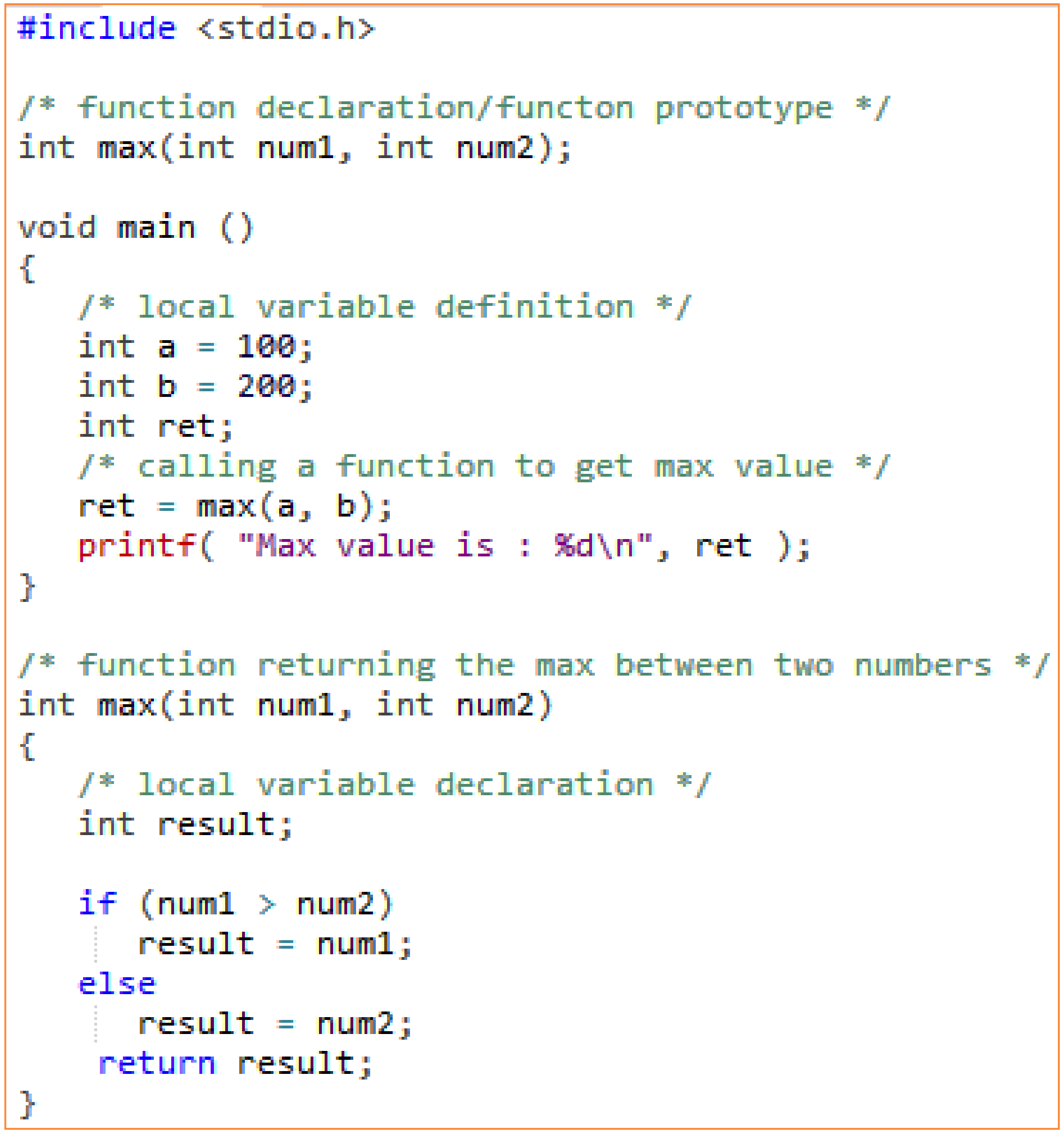
6. Function will return 16 to the calling function.(i.e main)
7. Returned value will be copied into variable.
8. printf will gets executed.
9. main function ends.
10. C program terminates.

CODE EXAMPLE: CALLING A FUNCTION



```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;
}

 void main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
}
```

CODE EXAMPLE: CALLING A FUNCTION



```c
#include <stdio.h>

/* function declaration/functon prototype */
int max(int num1, int num2);

void main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
}

/* 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;
}
```

A C program with function *declaration*/ function *prototype*.

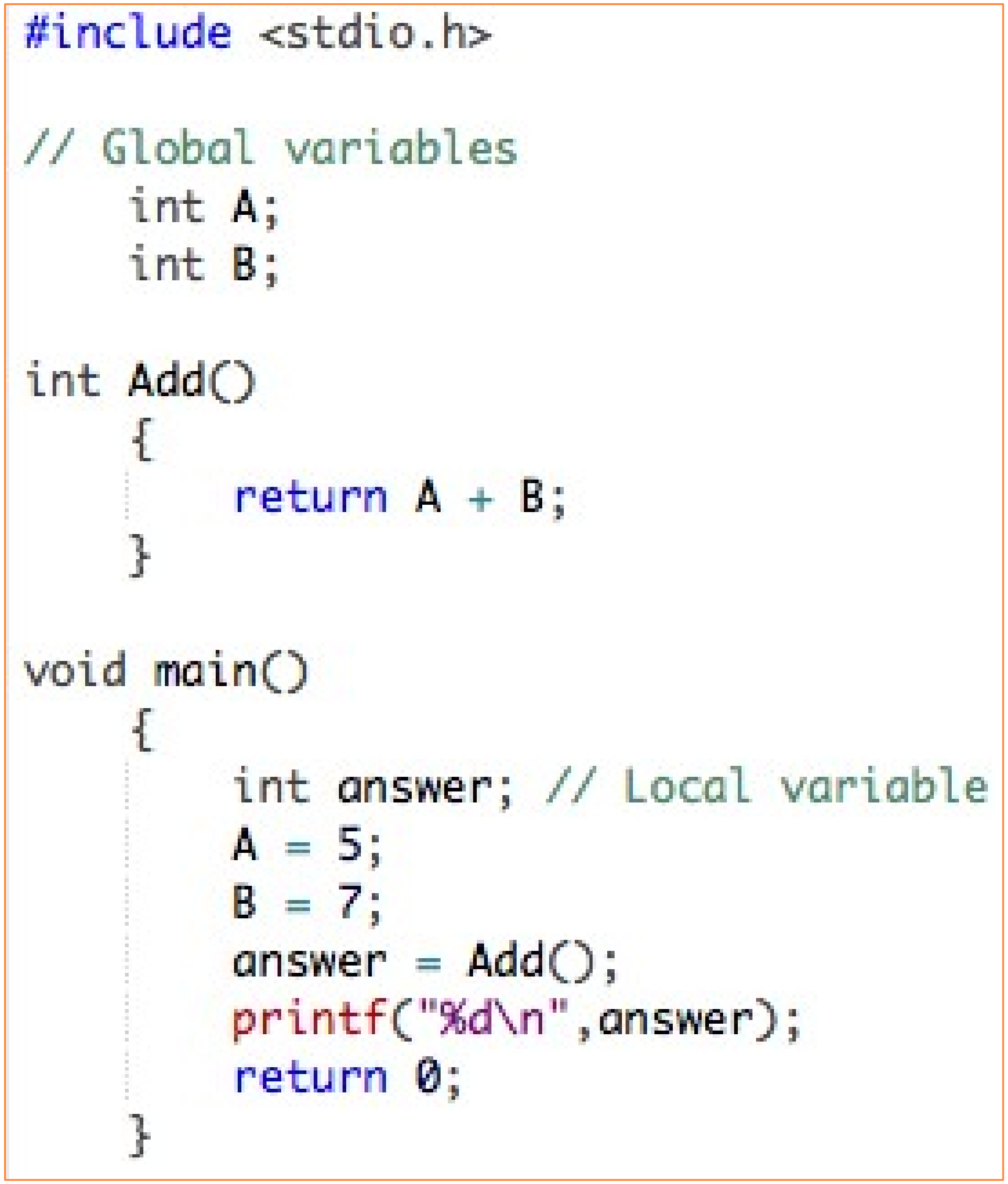
FUNCTION ARGUMENTS

- C functions can accept an unlimited number of parameters.
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the ***formal parameters*** of the function.
- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

GLOBAL AND LOCAL VARIABLES

- Local variable:
  - A local variable is a variable that is declared inside a function.
  - A local variable can only be used in the function where it is declared.
- Global variable:
  - A global variable is a variable that is declared outside **all** functions.
  - A global variable can be used in all functions.
- See the following example(see the next slide)

GLOBAL AND LOCAL VARIABLES



```c
#include <stdio.h>

// Global variables
    int A;
    int B;

int Add()
    {
        return A + B;
    }

void main()
    {
        int answer; // Local variable
        A = 5;
        B = 7;
        answer = Add();
        printf("%d\n",answer);
        return 0;
    }
```

- As you can see two global variables are declared, **A** and **B**. These variables can be used in main() and Add().
- The local variable **answer** can only be used in main().

TYPE OF FUNCTION CALL

While calling a function, there are two ways that arguments can be passed to a function:

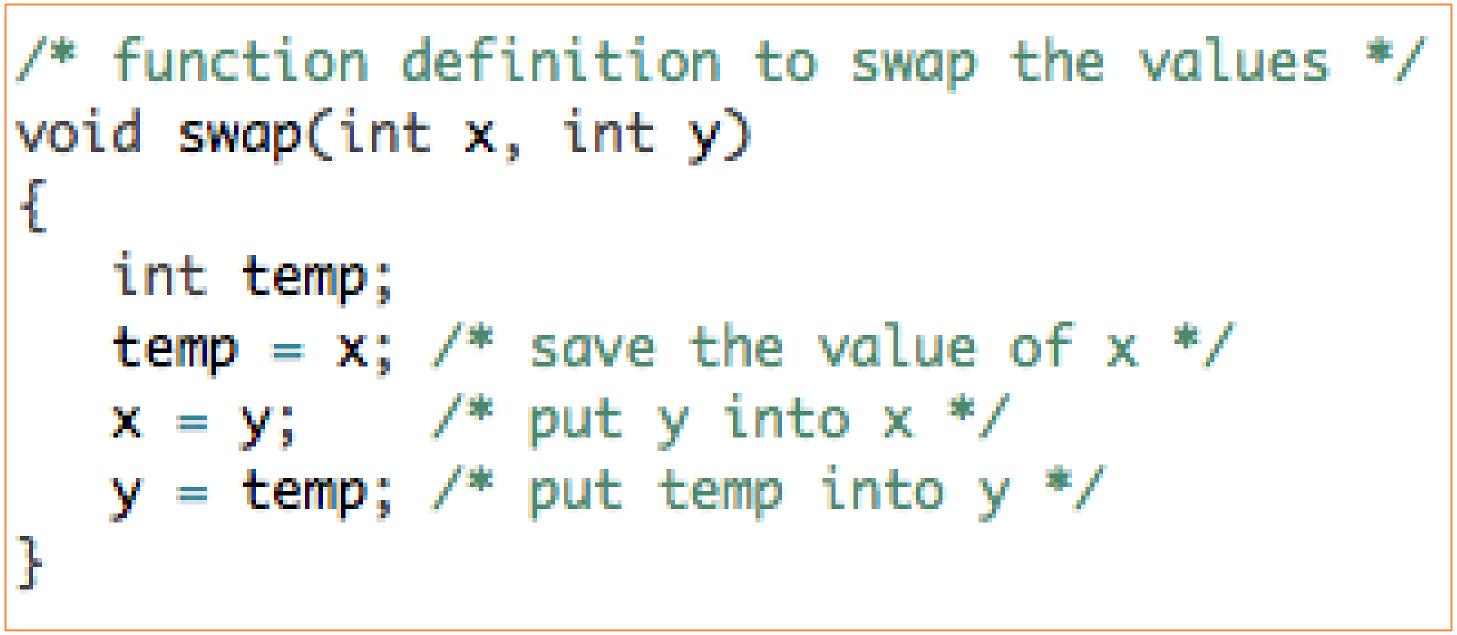
Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal 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.

CALL BY VALUE

- The **call by value** method of passing arguments to a function 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.
- By default, C programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

CALL BY VALUE

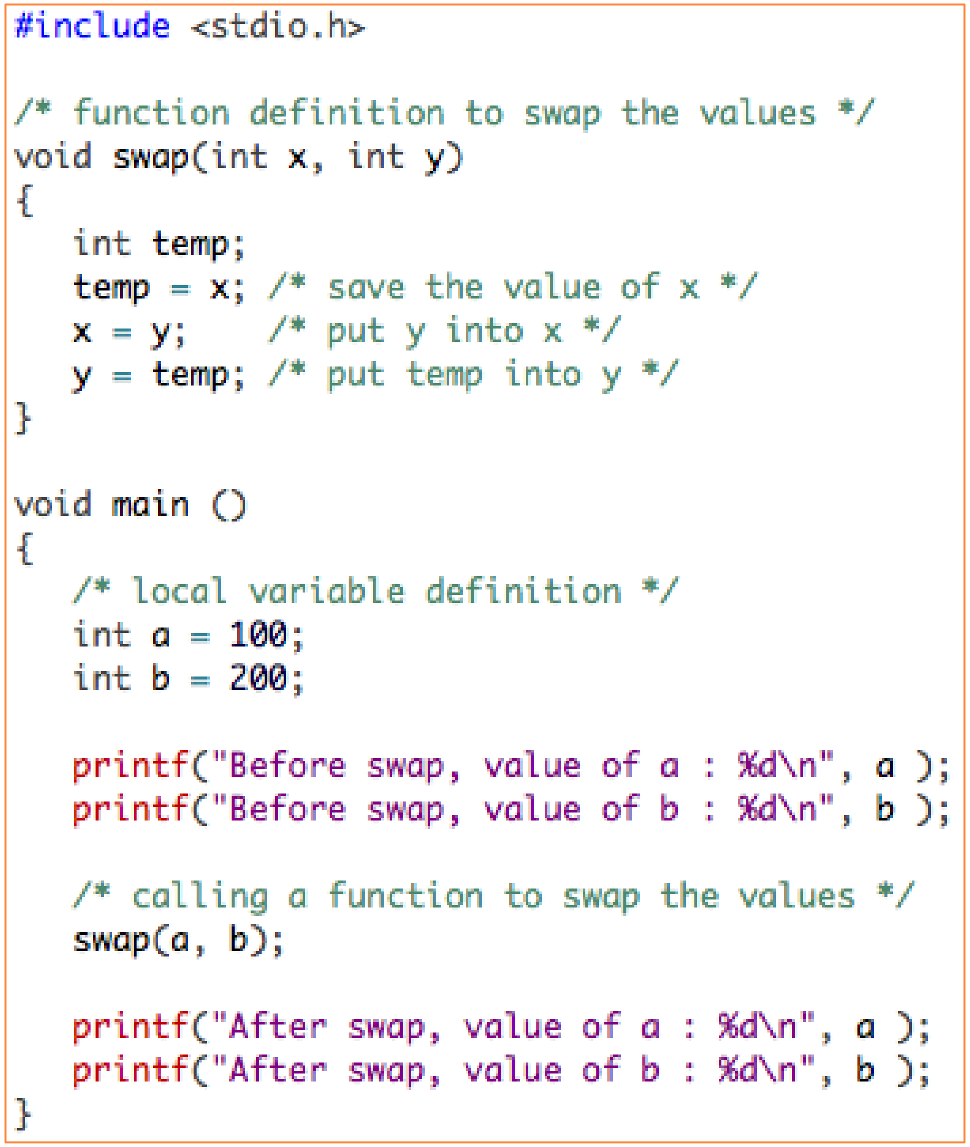
- Consider the function **swap()** definition as follows.



```
/* function definition to swap the values */
void swap(int x, int y)
{
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
}
```

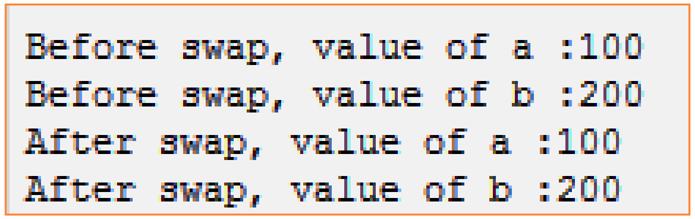
CALL BY VALUE

- Now, let us call the function **swap()** by passing actual values as in the following example:



```c
#include <stdio.h>

/* function definition to swap the values */
void swap(int x, int y)
{
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
}

void main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
}
```

Output:



```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

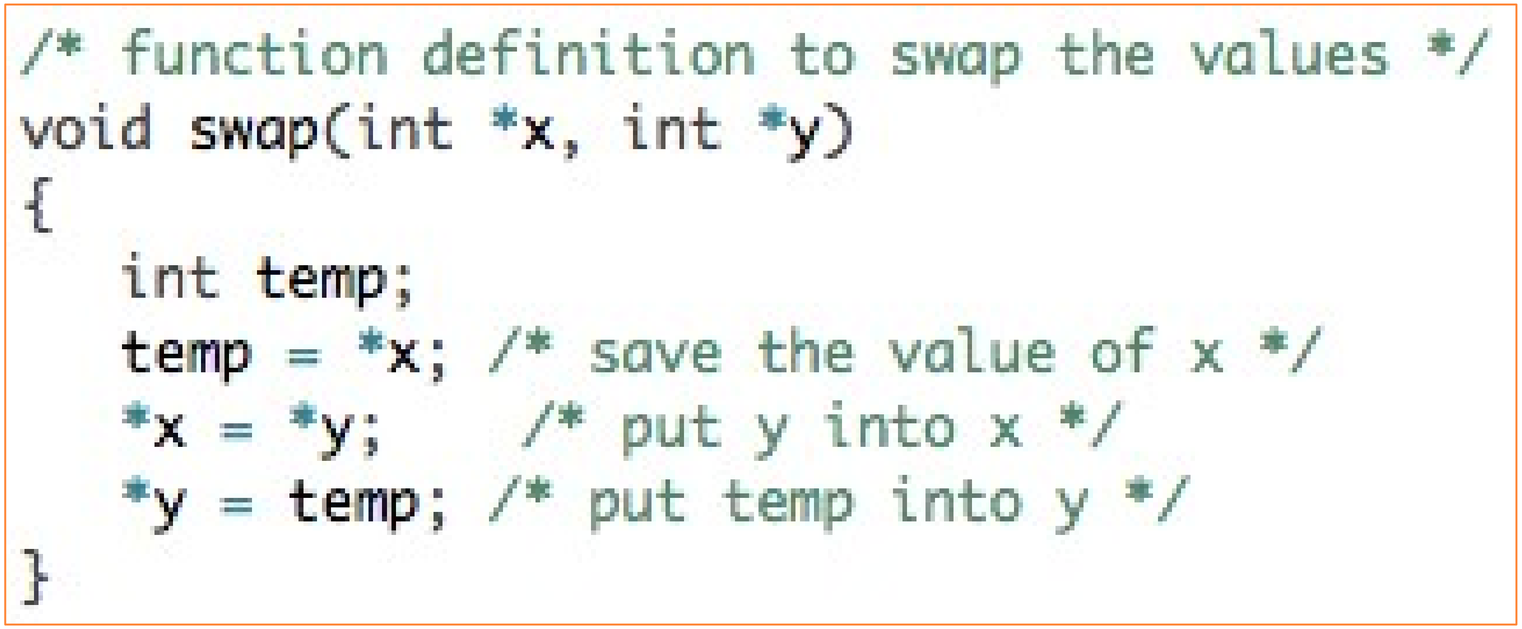
The output shows that there is no change in the values though they had been changed inside the function.

CALL BY REFERENCE

- The **call by reference** method of passing arguments to a function 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 passed argument.
- To pass the value by call by reference, argument pointers are passed to the functions.

FUNCTION CALL BY REFERENCE

- To pass the value by reference, argument pointers are passed to the functions just like any other value.
- So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.



```
/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value of x */
    *x = *y;    /* put y into x */
    *y = temp; /* put temp into y */
}
```

CALL BY REFERENCE

Let us call the function **swap()** by passing values by reference as in the following example:

```
#include <stdio.h>

/* function definition to swap the values */
void swap(int *x, int *y)
{
   int temp;
   temp = *x; /* save the value of x */
   *x = *y;    /* put y into x */
   *y = temp; /* put temp into y */
}

void main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
    * &a indicates pointer to a ie. address of variable a and
    * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
}
```

Output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Which shows that the change has reflected outside of the function as well unlike call by value where changes does not reflect outside of the function.

RECURSION

- Recursion is the process of repeating items in a self-similar way. Same applies in programming languages as well where if a programming allows you to call a function inside the same function that is called recursive call of the function as follows.

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

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

- The C programming language supports recursion, i.e., a function to call itself.
- But while using recursion, programmers need to be careful to define an exit condition (base condition) from the function, otherwise it will go in infinite loop.
- Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

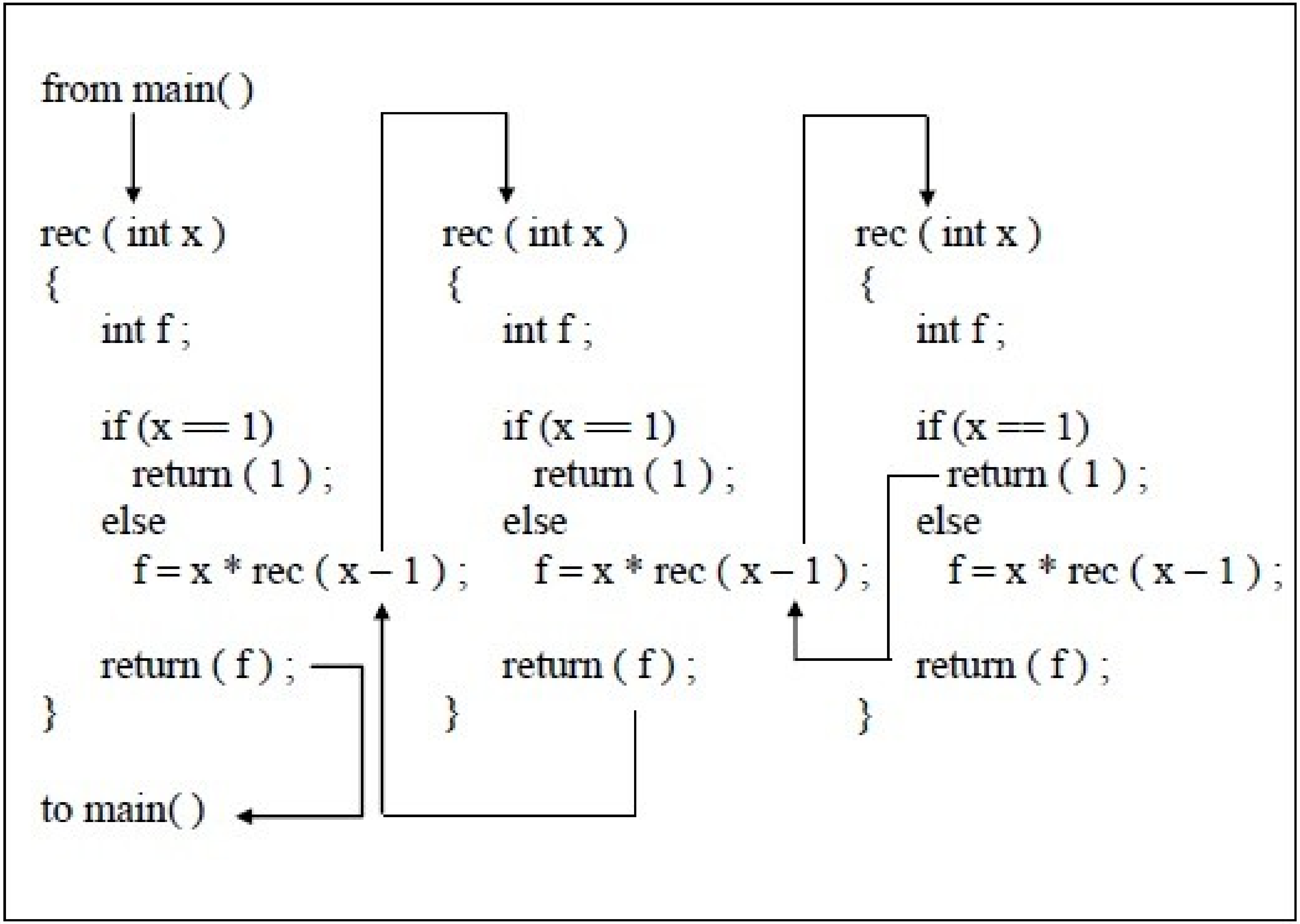
NUMBER FACTORIAL

- Following is an example, which calculates factorial for a given number using a recursive function:

```
#include <stdio.h>

int factorial(unsigned int i)
{
   if(i <= 1)
   {
      return 1;
   }
   return i * factorial(i - 1);
}
int  main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

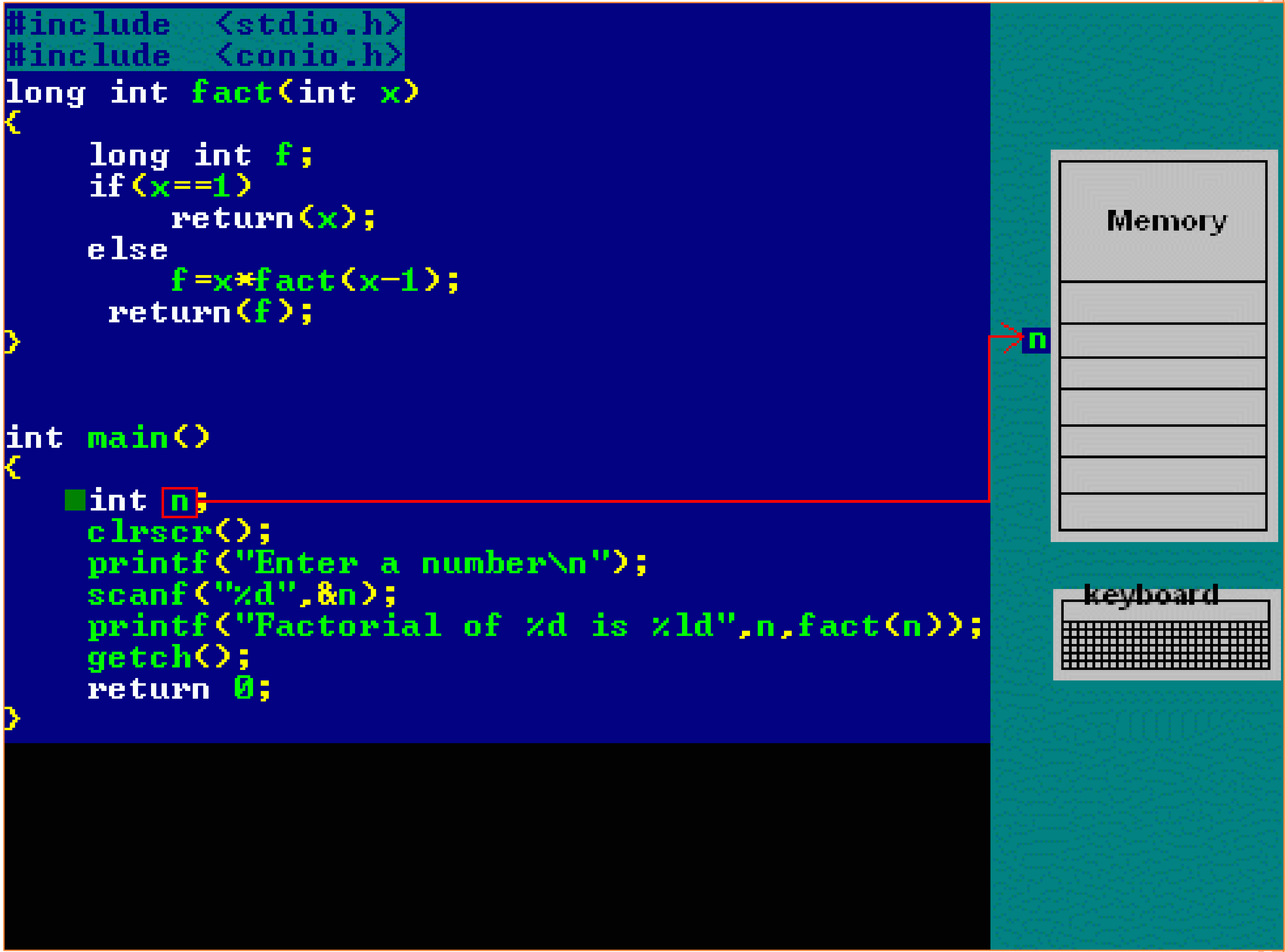
```
Factorial of 15 is 2004310016
```



```
rec ( int x )
{
    int f ;

    if (x == 1)
      return ( 1 ) ;
    else
      f = x * rec ( x – 1 ) ;

    return ( f ) ;
}
```

```
rec ( int x )
{
    int f ;

    if (x == 1)
      return ( 1 ) ;
    else
      f = x * rec ( x – 1 ) ;

    return ( f ) ;
}
```

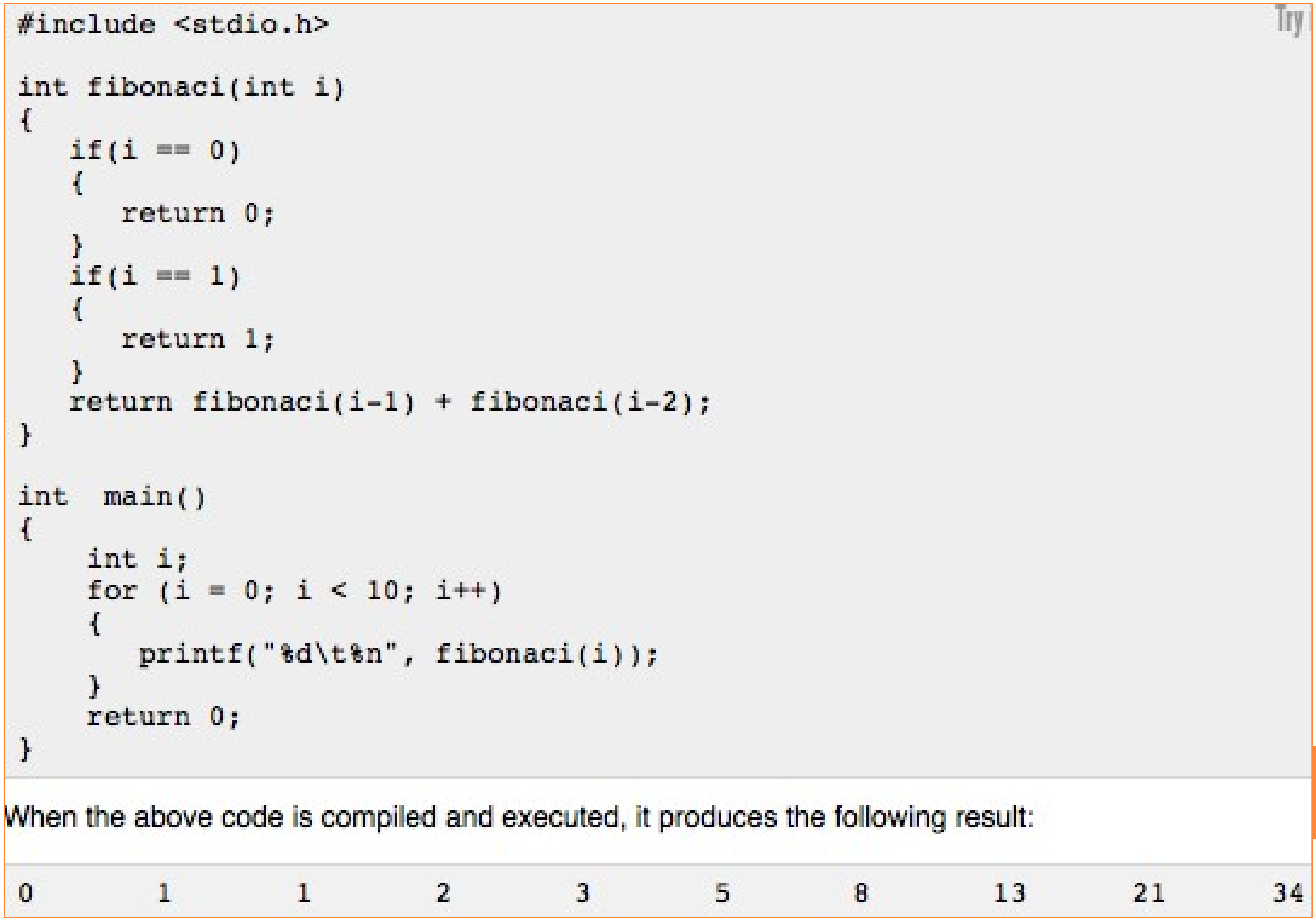
```
rec ( int x )
{
    int f ;

    if (x == 1)
      return ( 1 ) ;
    else
      f = x * rec ( x – 1 ) ;

    return ( f ) ;
}
```



```
#include  <stdio.h>
#include  <conio.h>
long int fact(int x)
{
    long int f;
    if(x==1)
        return(x);
    else
        f=x*fact(x-1);
     return(f);
}


int main()
{
   int n;
    clrscr();
    printf("Enter a number\n");
    scanf("%d",&n);
    printf("Factorial of %d is %ld",n,fact(n));
    getch();
    return 0;
}
```

FIBONACCI SERIES

- Following is another example, which generates Fibonacci series for a given number using a recursive function:



```
#include <stdio.h>

int fibonaci(int i)
{
   if(i == 0)
   {
      return 0;
   }
   if(i == 1)
   {
      return 1;
   }
   return fibonaci(i-1) + fibonaci(i-2);
}

int  main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
       printf("%d\t%n", fibonaci(i));
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
0       1       1       2       3       5       8       13      21      34
```