# FUNCTIONS

Module 3

# WHAT IS FUNCTION?

- A function is a self-contained block of statements that perform a coherent task of some kind.
- A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- 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 is such that 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, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.
- A function can also be referred as a method or a sub-routine or a procedure, etc.

# WHY WE USE FUNCTION

(a) Writing functions avoids rewriting the same code over and over.

(b) By using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently.

# DEFINING A FUNCTION

- The general form of a function definition in C programming language is as follows −

Syntax

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

- A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a 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. 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.
- **Function Body** − The function body contains a collection of statements that define what the function does.

# EXAMPLE

```
# include<stdio.h>
void message( ) ; /* function prototype declaration */
int main( )
 {
      message( ) ; /* function call */
      printf (“This is main function \n") ;
      return 0 ;
 }
void message( ) /* function definition */
 {
      printf (“This is user define function.\n") ;
}
```

OUTPUT
This is user define function
This is main function

Here, we have defined two functions—main( ) and message( ).
In fact, we have used the word message at three places in the program. Let us understand the meaning of each

- The first is the function prototype declaration and is written as:

```
void message( ) ;
```

This prototype declaration indicates that message( ) is a function which after completing its execution does not return any value.

This ‘does not return any value’ is indicated using the keyword void. It is necessary to mention the prototype of every function that we intend to define in the program.

- The second usage of message is…

```
void message( )
{
    printf ( "Smile, and the world smiles with you...\n" ) ;
```

} This is the function definition. In this definition right now we are having only printf( ), but we can also use if, for, while, switch, etc., within this function definition.

- The third usage is…

```
message( ) ;
```

Here the function message( ) is being called by main( ).

What do we mean when we say that main( ) ‘calls’ the function message( )? We mean that the control passes to the function message( ). The activity of main( ) is temporarily suspended; it falls asleep while the message( ) function wakes up and goes to work. When the message( ) function runs out of statements to execute, the control returns to main( ), which comes to life again and begins executing its code at the exact point where it left off. Thus**, main( ) becomes the ‘calling’ function**, whereas **message( ) becomes the ‘called’ function.**

# EXAMPLE

```
# include<stdio.h>
void italy( ) ;
void brazil( ) ;
void argentina( ) ;
int main( )
 {
       printf ( "I am in main\n" ) ;
       italy( ) ;
      brazil( ) ;
       argentina( ) ;
       return 0 ;
}
void italy( )
 {
      printf ( "I am in italy\n" ) ;
 }
void brazil( )
{
      printf ( "I am in brazil\n" ) ;
}
void argentina( )
 {
      printf ( "I am in argentina\n" ) ;
 }
```

# CONCLUSION

- A C program is a collection of one or more functions.
- If a C program contains only one function, it must be main( ).
- If a C program contains more than one function, then one (and only one) of these functions must be main( ), because program execution always begins with main( ).
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in main( ).
- After each function has done its thing, control returns to main( ). When main( ) runs out of statements and function calls, the program ends.

- As we have noted earlier, the program execution always begins with main( ).
- Except for this fact, all C functions enjoy a state of perfect equality. No precedence, no priorities, nobody is nobody's boss.
- One function can call another function it has already called but has in the meantime left temporarily in order to call a third function which will sometime later call the function that has called it.

# EXAMPLE

```
# include<stdio.h>
void italy( ) ;
void brazil( ) ;
void argentina( ) ;
int main( )
{
      printf ( "I am in main\n" ) ;
      italy( ) ;
      printf ( "I am finally back in main\n" ) ;
       return 0 ;
}
```

```
void italy( )
 {
       printf ( "I am in italy\n" ) ;
      brazil( ) ;
      printf ( "I am back in italy\n" ) ;
 }
 void brazil( )
{
       printf ( "I am in brazil\n" ) ;
      argentina( ) ;
}
void argentina( )
{
       printf ( "I am in argentina\n" )
}
```

# OUTPUT

I am in main

I am in Italy

I am in brazil

I am in argentina

I am back in italy

I am finally back in main

( C )Any function can be called from any other function. Even main( ) can be called from other functions. For example,

```
 # include
void message( ) ;
int main( )
{
message( ) ;
return 0 ;
}
void message( )
 {
       printf ( "Can't imagine life without C\n" ) ;
       main( ) ;
}
```

(d) A function can be called any number of times. For example,

```
 #include<stdio.h>
void message( ) ;
 int main( )
{
        message( ) ;
       message( ) ;
       return 0 ;
}
void message( )
{
       printf ( "Jewel Thief!!\n" ) ;
 }
```

(e) The order in which the functions are defined in a program and the order in which they get called need not necessarily be same.

For example,

```
# include<stdio.h>
 void message1( ) ;
void message2( ) ;
int main( )
 {
       message1( ) ;
      message2( ) ;
      return 0 ;
}
void message2( )
{
       printf ( "But the butter was bitter\n" ) ;
 }
void message1( )
{
       printf ( "Mary bought some butter\n" ) ;
}
```

Here, even though message1( ) is getting called before message2( ), still, message1( ) has been defined after message2( ). However, it is advisable to define the functions in the same order in which they are called. This makes the program easier to understand.

(f) A function can call itself. Such a process is called ‘recursion’.

(g) A function can be called from another function, but a function cannot be defined in another function. Thus, the following program code would be wrong, since argentina( ) is being defined inside another function, main( ):

```
int main( )
{
    printf ( "I am in main\n" ) ;
    void argentina( )
    {
        printf ( "I am in argentina\n" ) ;
    }
}
```

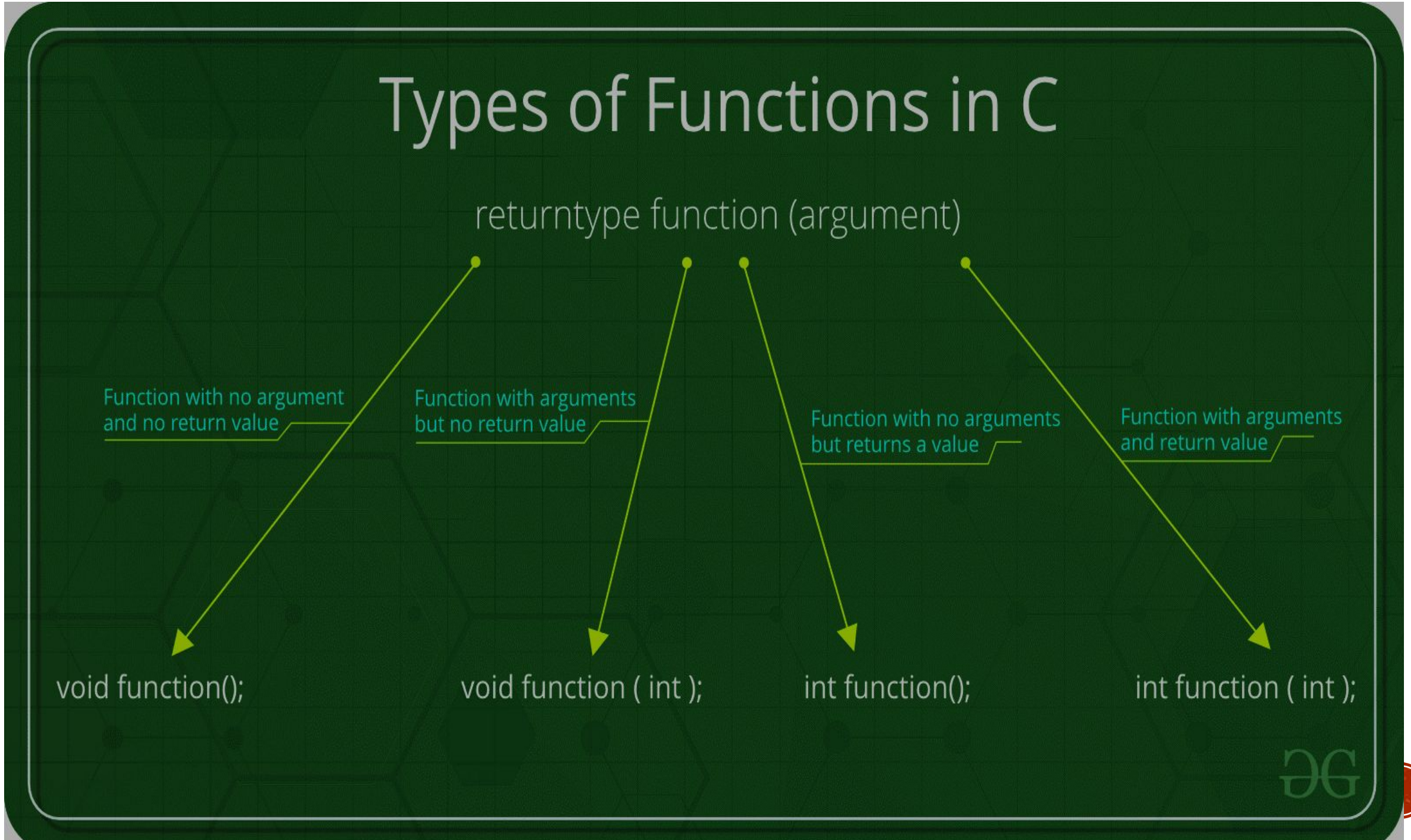
(h) There are basically two types of functions:

Library functions

Ex. printf( ), scanf( ), etc.

User-defined functions

Ex. argentina( ), brazil( ), etc. As the name suggests, library functions are nothing but commonly required functions grouped together and stored in a Library file on the disk. These library of functions come ready-made with development environments like Turbo C, Visual Studio, NetBeans, gcc, etc. The procedure for calling both types of functions is exactly same.


Types of Functions in C
returntype function (argument)
Function with no argument and no return value
Function with arguments but no return value
Function with no arguments but returns a value
Function with arguments and return value
void function();
void function ( int );
int function();
int function ( int );

# FUNCTION WITH NO ARGUMENT AND NO RETURN VALUE

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

**Function declaration :** void function();

**Function call :** function();

**Function definition :** void function() { statements; }

```c
// C code for function with no arguments and no
return value

#include <stdio.h>
void value(void);
void main()
{
   value();
}
void value(void)
{
   int a=19,b=20, sum;
   sum=a+b;
   printf(" The Sum is %d:", sum);
}
```

OUTPUT:
The Sum is 39:

# FUNCTION WITH ARGUMENTS BUT NO RETURN VALUE

When a function has arguments, it receive any data from the calling function but it returns no values.

Syntax :

**Function declaration :** void function ( int );

**Function call :** function( x );

**Function definition:** void function( int x )

```
{
    statements;
}
```

```c
// C code for function with argument but no return value
#include <stdio.h>
void function(int, int);
int main()
{
    int a = 20,b=19;
    /*int a,b;
    printf("Enter two nos");
    scanf("%d %d",&a,&b);*/
    function(a, b);
    return 0;
}
void function(int a, int b)
{
    int sum;
    sum=a+b;
    printf("\nThe Sum is %d\n",sum);
}
```

OUTPUT:
The Sum is 39

# FUNCTION WITH NO ARGUMENTS BUT RETURNS A VALUE

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function.

Syntax :

**Function declaration :** int function();

**Function call :** function();

**Function definition :** int function()

```
{
    statements;
    return x;
}
```

```c
// C code for function with no arguments but have return value
#include <math.h>
#include <stdio.h>

int sum();
int main()
{
    int num;
    num = sum();
    printf("\nSum of two given values = %d", num);
    return 0;
}
int sum()
{
    int a = 50, b = 80, sum;
    sum = a + b;
    return sum;
}
```

OUTPUT:
Sum of two given values = 130

# FUNCTION WITH ARGUMENTS AND RETURN VALUE

Syntax :

**Function declaration :** int function ( int );

**Function call :** function( x );

**Function definition:** int function( int x )

```
{
    statements;
    return x;
}
```

```c
// C code for function with arguments and with return value
#include <stdio.h>
#include <string.h>
int function(int);
int main()
{
    int a = 20,b;
      b = function(a);
    printf("value of b is %d\n", b);
    return 0;
}
int function(int a)
{
    a = a + 10;
    return a;
}
```

OUTPUT:
value of b is 30

# PASSING VALUES BETWEEN FUNCTIONS

- The mechanism used to convey information to the function is the ‘argument’.
- You have unknowingly used the arguments in the printf( ) and scanf( ) functions; the format string and the list of variables used inside the parentheses in these functions are arguments.
- The arguments are sometimes also called ‘parameters’

# EXAMPLE

```
/* Sending and receiving values between functions
*/
 # include<stdio.h>
 int calsum ( int x, int y, int z ) ;
int main( )
{
       int a, b, c, sum ;
      printf ( "Enter any three numbers " ) ;
      scanf ( "%d %d %d", &a, &b, &c ) ;
      sum = calsum ( a, b, c ) ;
      printf ( "Sum = %d\n", sum ) ;
      return 0 ;
}
int calsum ( int x, int y, int z )
 {
      int d ;
      d = x + y + z ;
      return ( d ) ;
 }
```

# CONT…

There are a number of things to note about this program:

(a) In this program, from the function main( ), the values of a, b and c are passed on to the function calsum( ), by making a call to the function calsum( ) and mentioning a, b and c in the parentheses:

```
sum = calsum ( a, b, c ) ;
```

In the calsum( ) function these values get collected in three variables x, y and z: int calsum ( int x, int y, int z )

b) **The variables a, b and c are called 'actual arguments', whereas the variables x, y and z are called 'formal arguments'.**

Any number of arguments can be passed to a function being called. However, the type, order and number of the actual and formal arguments must always be same. Instead of using different variable names x, y and z, we could have used the same variable names a, b and c. But the compiler would still treat them as different variables since they are in different functions

# CONT..

| Actual Parameters | Formal Parameters |
|---|---|
| Actual Parameters used in the function call. | Formal Parameters used in the function header. |
| Data type not required. | Data type define must be required. |
| Parameters can be constant values or variable names. | Parameters can be handle as local variables. |
| Ex:- add(a,b);<br>**a and b are Actual parameters** | Ex:- int add(int a, int b)<br>{<br>//All function code<br>}<br>**a and b are Formal parameters** |
| The Actual parameters are the values that are passed to the function when it is invoked. | The Formal Parameters are the variables defined by the function that receives values when the function is called. |
| The actual parameters are passed by the calling function. | The formal parameters are in the called function. |

# CONT…

(c) Note the function prototype declaration of calsum( ).

Instead of the usual void, we are using int. This indicates that calsum( ) is going to return a value of the type int. It is not compulsory to use variable names in the prototype declaration. Hence we could as well have written the prototype as:

```
int calsum ( int, int, int ) ;
```

In the definition of calsum too, void has been replaced by int.

(d) In the earlier programs, the moment closing brace ( } ) of the called function was encountered, the control returned to the calling function. No separate return statement was necessary to send back the control. This approach is fine if the called function is not going to return any meaningful value to the calling function.

In the above program, however, we want to return the sum of x, y and z. Therefore, it is necessary to use the return statement.

The return statement serves two purposes:

(1) On executing the return statement, it immediately transfers the control back to the calling function.

(2) It returns the value present in the parentheses after return, to the calling function. In the above program, the value of sum of three numbers is being returned.

# CONT..

(e) There is no restriction on the number of return statements that may be present in a function. Also, the return statement need not always be present at the end of the called function. The following program illustrates these facts:

```
int fun( )
{
    int n ;
    printf ( "Enter any number " ) ;
    scanf ( "%d", &n ) ;
    if ( n >= 10 && n <= 90 )
    return ( n ) ;
    else
    return ( n + 32 ) ;
}
```

In this function, different return statements will be executed depending on whether n is between 10 and 90 or not.

# CONT..

(f) Whenever the control returns from a function, the sum being returned is collected in the calling function by assigning the called function to some variable.

For example, sum = calsum ( a, b, c ) ;

(g) All the following are valid return statements.

```
return ( a ) ;

return ( 23 ) ;

return ;
```

In the last statement, a garbage value is returned to the calling function since we are not returning any specific value. Note that, in this case, the parentheses after return are dropped. In the other return statements too, the parentheses can be dropped.

# CONT…

(h) A function can return only one value at a time. Thus, the following statements are invalid:

```
return ( a, b ) ;
return ( x, 12 ) ;
```

# CONT..

(i) If the value of a formal argument is changed in the called function, the corresponding change does not take place in the calling function. For example

```
# include<stdio.h>
 void fun ( int ) ;
int main( )
{
      int a = 30 ;
      fun ( a ) ;
      printf ( "%d\n", a ) ;
      return 0 ;
}
```

```
void fun ( int b )
 {
       b = 60 ;
      printf ( "%d\n", b ) ;
 }
```

The output of the above program would be: 60 30

Thus, even though the value of b is changed in fun( ), the value of a in main( ) remains unchanged. This means that when values are passed to a called function, the values present in actual arguments are not physically moved to the formal arguments; just a photocopy of values in actual argument is made into formal arguments.

# SCOPE RULE OF FUNCTIONS

Example

```
# include void display ( int ) ;
int main( )
{
        int i = 20 ;
       display ( i ) ;
        return 0 ;
 }
void display ( int j )
{       int k = 35;
        printf ( "%d\n", j ) ;
       printf ( "%d\n", k ) ;
 }
```

OUTPUT: 20,35

In this program, is it necessary to pass the value of the variable i to the function display( )?

Will it not become automatically available to the function display( )? No. Because, by default, the scope of a variable is local to the function in which it is defined. The presence of i is known only to the function main( ) and not to any other function.

Similarly, the variable k is local to the function display( ) and hence it is not available to main( ). That is why to make the value of i available to display( ), we have to explicitly pass it to display( ).

Likewise, if we want k to be available to main( ), we will have to return it to main( ) using the return statement. In general, we can say that the scope of a variable is local to the function in which it is defined.

# ORDER OF PASSING ARGUMENTS

Consider the following function call:

```
fun (a, b, c, d ) ;
```

In this call, it doesn’t matter whether the arguments are passed from left to right or from right to left.

However, in some function calls, the order of passing arguments becomes an important consideration. For example:

```
int a = 1 ;
printf ( "%d %d %d\n", a, ++a, a++ ) ;
```

output 1 2 2.

This however is not the case. Surprisingly, it outputs 1 3 3

# CONT…

- This is because, **in C during a function call the arguments are passed from right to left.**
- Firstly 1 is passed through the expression a++ and then a is incremented to 2. Then result of ++a is passed. That is, a is incremented to 3 and then passed.
- Finally, latest value of a, i.e., 3, is passed. Thus in right to left order, 1, 3, 3 get passed. Once printf( ) collects them, it prints them in the order in which we have asked it to get them printed (and not the order in which they were passed).
- Thus 3 3 1 gets printed

# USING LIBRARY FUNCTIONS

- Consider the following program:

```
# include <stdio.h>
# include<conio.h>
 int main( )
{
        float a = 0.5 ;
        float w, x, y, z ;
        w = sin ( a ) ;
        x = cos ( a ) ;
        y = tan ( a ) ;
        z = pow ( a, 2 ) ;
        printf ( "%f %f %f %f\n", w, x, y, z ) ;
        return 0 ;
}
```

- Here we have called four standard library functions—sin( ), cos( ), tan( ) and pow( ). As we know, before calling any function, we must declare its prototype. This helps the compiler in checking whether the values being passed and returned are as per the prototype declaration.
- Prototypes of functions sin( ), cos( ), tan( ) and pow( ) are declared in the file ‘math.h’. To make these prototypes available to our program we need to include the file ‘math.h’. You can even open this file and look at the prototypes.

# CALL BY VALUE

- In call by value, **original value can not be changed** or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location.
- If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller method such as main().

# EXAMPLE

```c
//call by value
include<stdio.h>
void swapx(int x, int y);
// Main function
int main()
{
    int a = 10, b = 20;
    // Pass by Values
    printf("a= %d b= %d \n", a, b);
     swapx(a, b);
    printf("a= %d b= %d \n", a, b);
    return 0;
}
```

```c
void swapx(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
 printf("a= %d b= %d \n", a, b);
}
```

OUTPUT:
10 20
20 10
10 20

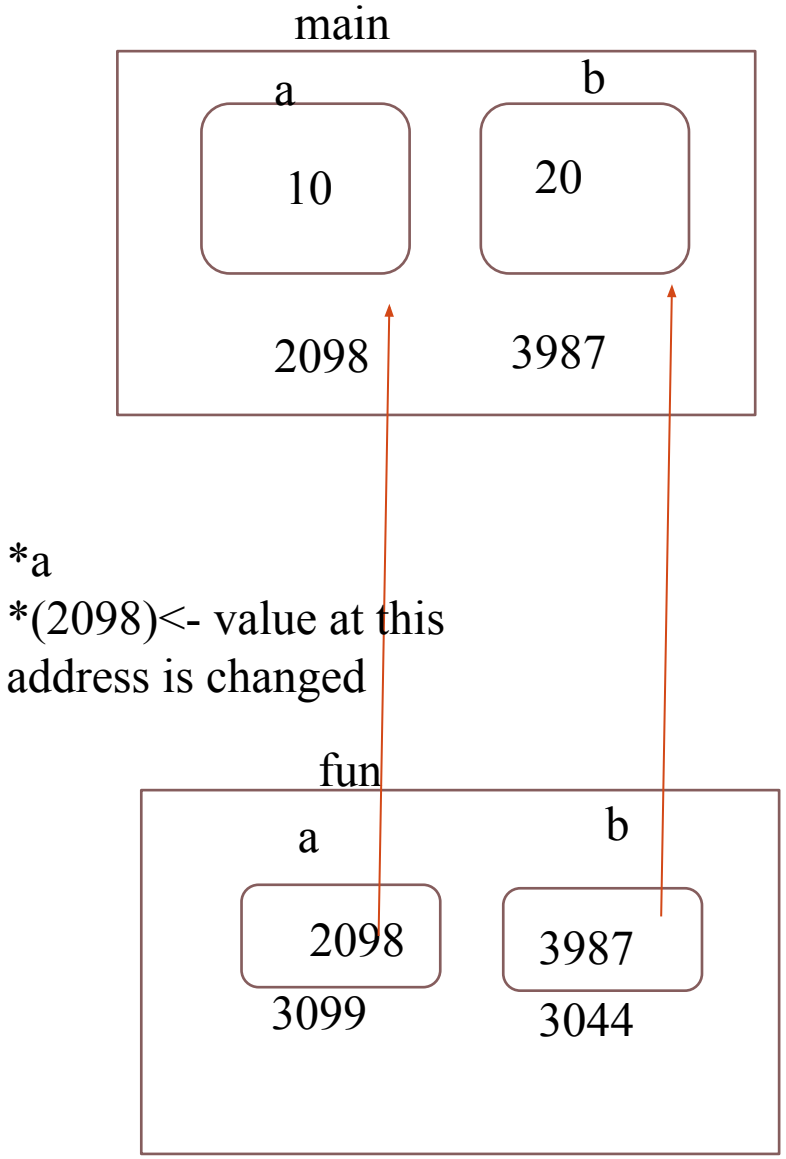
# CALL BY REFERENCE

- In call by reference, **original value is changed** or modified because we pass reference (address).
- Here, address of the value is passed in the function, so actual and formal arguments shares the same address space.
- Hence, any value changed inside the function, is reflected inside as well as outside the function.

# EXAMPLE

```c
#include<stdio.h>
void swapx(int*, int*);
int main()
{
    int a = 10, b = 20;
    printf("a=%d b=%d\n", a, b);
    swapx(&a, &b);
    printf("a=%d b=%d\n", a, b);
    return 0;
}
void swapx(int* a, int* b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
    printf("a=%d b=%d\n", *a, *b);
}
```

a=10 b=20
a=20 b=10
a=20 b=10



| Parameters | Call by value | Call by reference |
| --- | --- | --- |
| Definition | While calling a function, when you pass values by copying variables, it is known as “Call By Values.” | While calling a function, in programming language instead of copying the values of variables, the address of the variables is used it is known as “Call By References. |
| Arguments | In this method, a copy of the variable is passed. | In this method, a variable itself is passed. |
| Effect | Changes made in a copy of variable never modify the value of variable outside the function. | Change in the variable also affects the value of the variable outside the function. |
| Alteration of value | Does not allow you to make any changes in the actual variables. | Allows you to make changes in the values of variables by using function calls. |
| Passing of variable | Values of variables are passed using a straightforward method. | Pointer variables are required to store the address of variables. |

| Parameters | Call by value | Call by reference |
| --- | --- | --- |
| Value modification | Original value not modified. | The original value is modified. |
| Memory Location | Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in the same memory location |
| Safety | Actual arguments remain safe as they cannot be modified accidentally. | Actual arguments are not Safe. They can be accidentally modified, so you need to handle arguments operations carefully. |
| Default | Default in many programming languages like C++.PHP. Visual Basic NET, and C#. | It is supported by most programming languages like JAVA, but not as default. |

# POINTER NOTATION

- Consider the declaration,

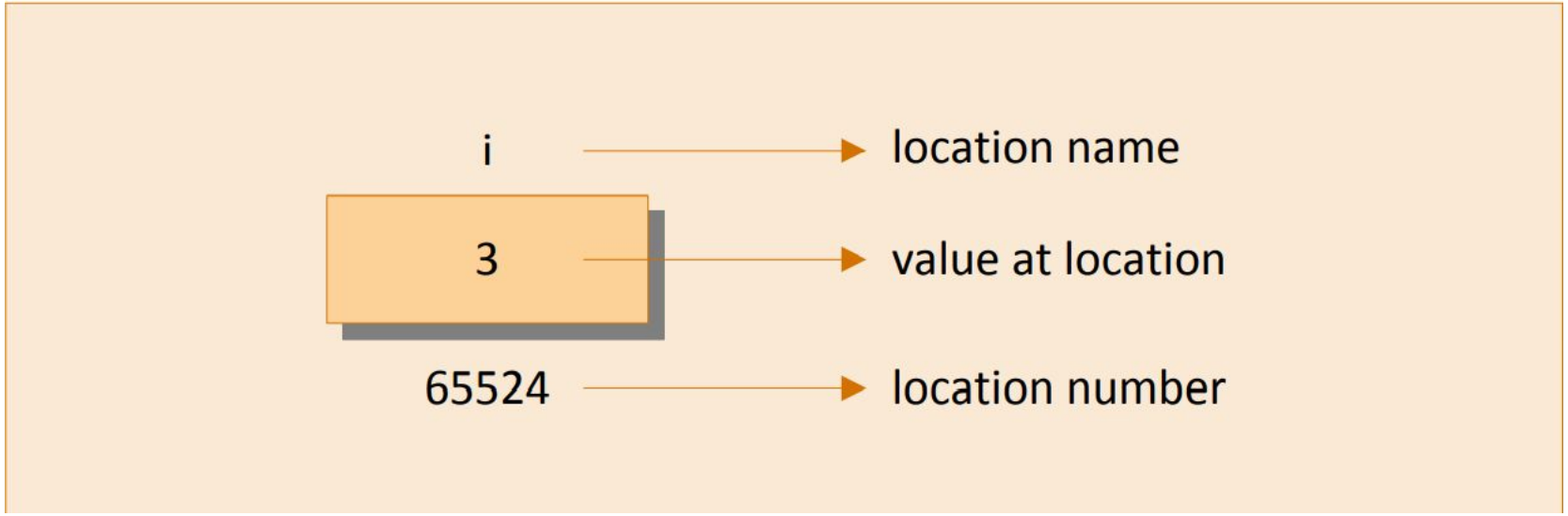
  int i = 3 ;

  This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name i with this memory location.

(c) Store the value 3 at this location.



```
# include int main( )
{
        int i = 3 ;
       printf ( "Address of i = %u\n", &i ) ;
       printf ( "Value of i = %d\n", i ) ;
       return 0 ;
}
```

OUTPUT:

Address of i = 65524

Value of i = 3

- The other pointer operator available in C is '*', called '**value at address**' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.
- Observe carefully the output of the following program:

```
# include<stdio.h>
 int main( )
 {
        int i = 3 ;
       printf ( "Address of i = %u\n", &i ) ;
       printf ( "Value of i = %d\n", i ) ;
       printf ( "Value of i = %d\n", *( &i ) ) ;
       return 0 ;
 }
```

The output of the above program would be:

Address of i = 65524

Value of i = 3

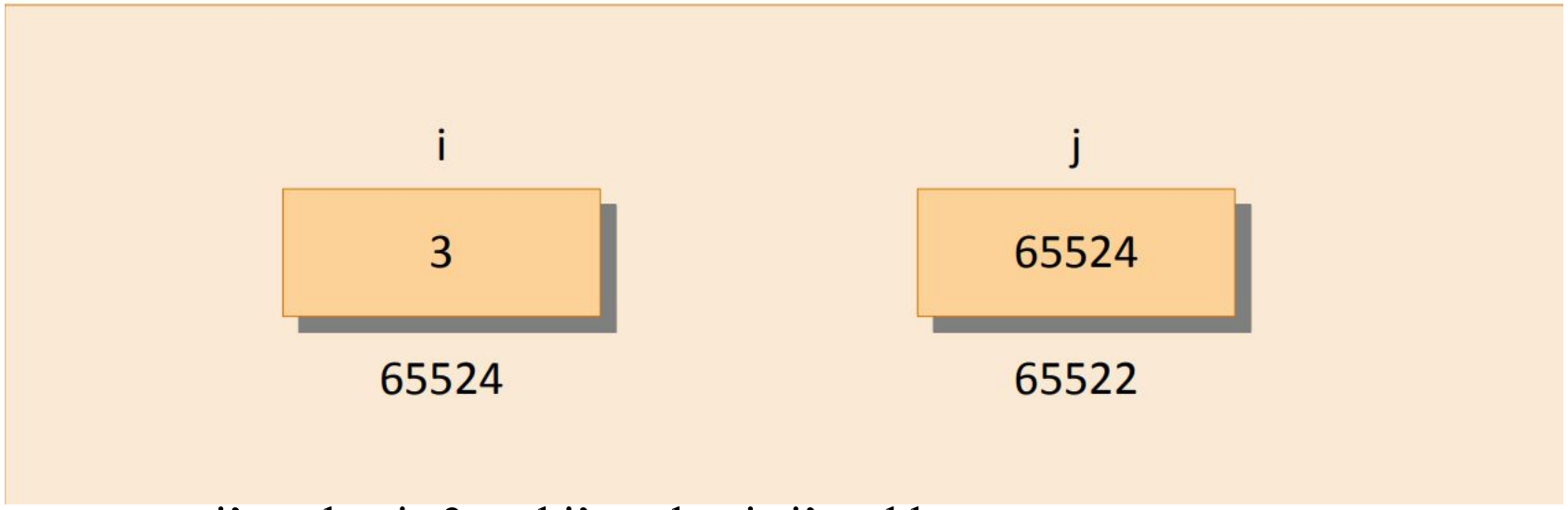
Value of i = 3

Note that printing the value of *( &i ) is same as printing the value of i.

- The expression **&i** gives the address of the variable i. This address can be collected in a variable, by saying,

  j = &i ;

- But remember that j is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (i in this case). Since j is a variable, the compiler must provide it space in the memory. Once again, the memory map shown in Figure would illustrate the contents of i and j.




- As you can see, i's value is 3 and j's value is i's address.

- But wait, we can't use j in a program without declaring it. And since j is a variable that contains the address of i, it is declared as,

  int *j ;

- This declaration tells the compiler that j will be used to store the address of an integer value. In other words, j points to an integer. How do we justify the usage of * in the declaration,

  int *j ;

- Let us go by the meaning of *. It stands for **'value at address'**. Thus, int *j would mean, the value at the address contained in j is an int.

# RECURSION

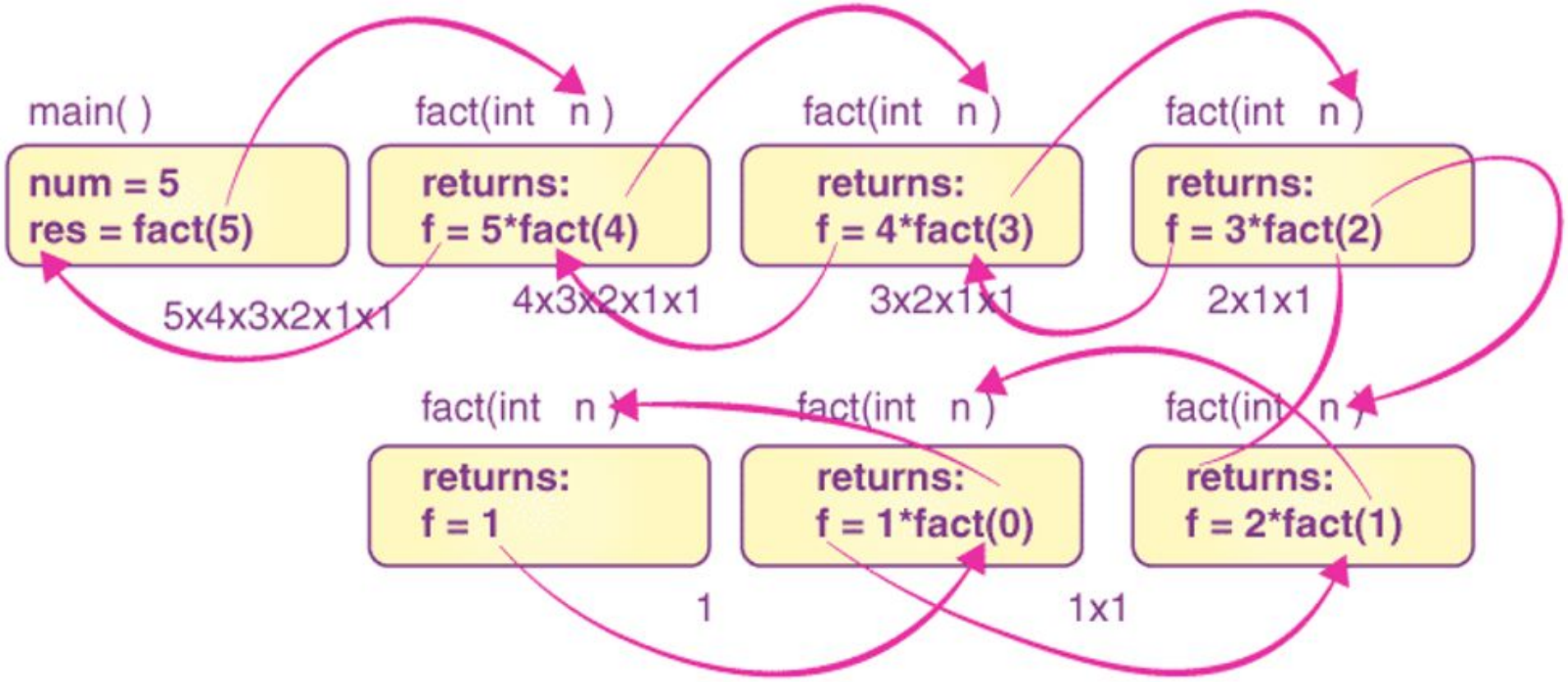
- In C, it is possible for the functions to call themselves. A function is called ‘recursive’ if a statement within the body of a function calls the same function. Sometimes called ‘circular definition’, recursion is thus the process of defining something in terms of itself.

```c
//factorial of number using function
# include <stdio.h>
int factorial ( int ) ;
int main( )
{
int a, fact ;
printf ( "Enter any number " ) ;
scanf ( "%d", &a ) ;
fact = factorial ( a ) ;
printf ( "Factorial value = %d\n", fact )
;
return 0 ;
}
```

```c
int factorial ( int x )
{
int f = 1, i ;
for ( i = x ; i >= 1 ; i-- )
f = f * i ;
return ( f ) ;
}
```

```c
//Recursive version of the function
to calculate the

#include <stdio.h>
int rec ( int );
int main( )
{
int a, fact ;
printf( "Enter any number ");
scanf( "%d", &a );
fact = rec ( a );
printf( "Factorial value = %d\n",
fact);
return 0 ;
}
```

```c
int rec ( int x )
{
int f ;
if ( x == 1 )
return ( 1 ) ;
else
f = x * rec ( x - 1 ) ;
return ( f ) ;
}
```


main( )
num = 5
res = fact(5)
fact(int n )
returns:
f = 5*fact(4)
fact(int n )
returns:
f = 4*fact(3)
fact(int n )
returns:
f = 3*fact(2)
fact(int n )
returns:
f = 2*fact(1)
fact(int n )
returns:
f = 1*fact(0)
fact(int n )
returns:
f = 1
5x4x3x2x1x1
4x3x2x1x1
3x2x1x1
2x1x1
1x1
1

- In the first run when the number entered through scanf( ) is 1, let us see what action does rec( ) take. The value of a (i.e., 1) is copied into x. Since x turns out to be 1, the condition if ( x == 1 ) is satisfied and hence 1 (which indeed is the value of 1 factorial) is returned through the return statement.
- When the number entered through scanf( ) is 2, the ( x == 1 ) test fails, so we reach the statement,

```
f = x * rec ( x - 1 ) ;
```

And here is where we meet recursion.

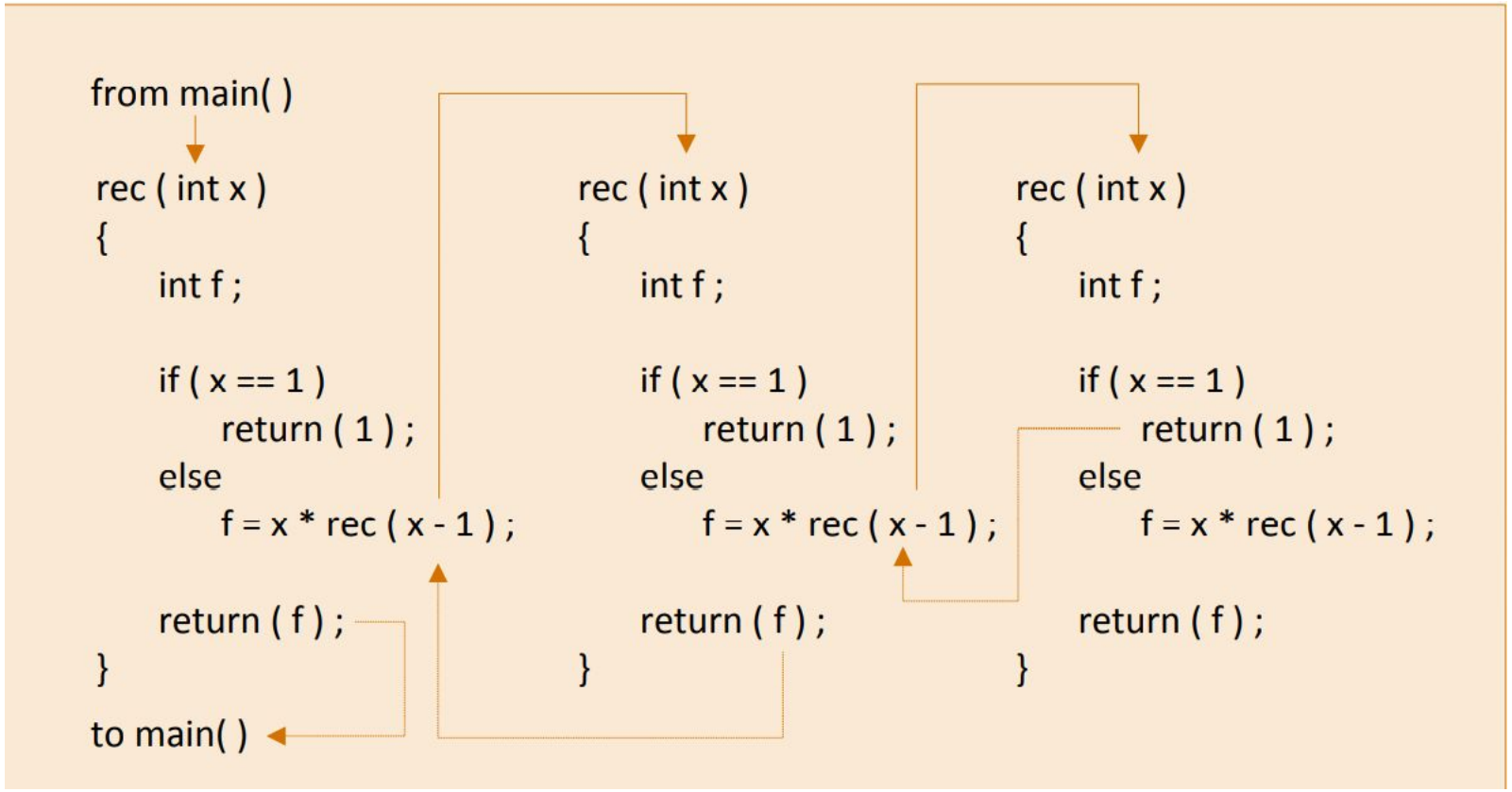
How do we handle the expression x * rec ( x - 1 )? We multiply x by rec ( x - 1 ). Since the current value of x is 2, it is same as saying that we must calculate the value (2 * rec ( 1 )). We know that the value returned by rec ( 1 ) is 1, so the expression reduces to (2 * 1), or simply 2. Thus the statement,

```
x * rec ( x - 1 ) ;
```

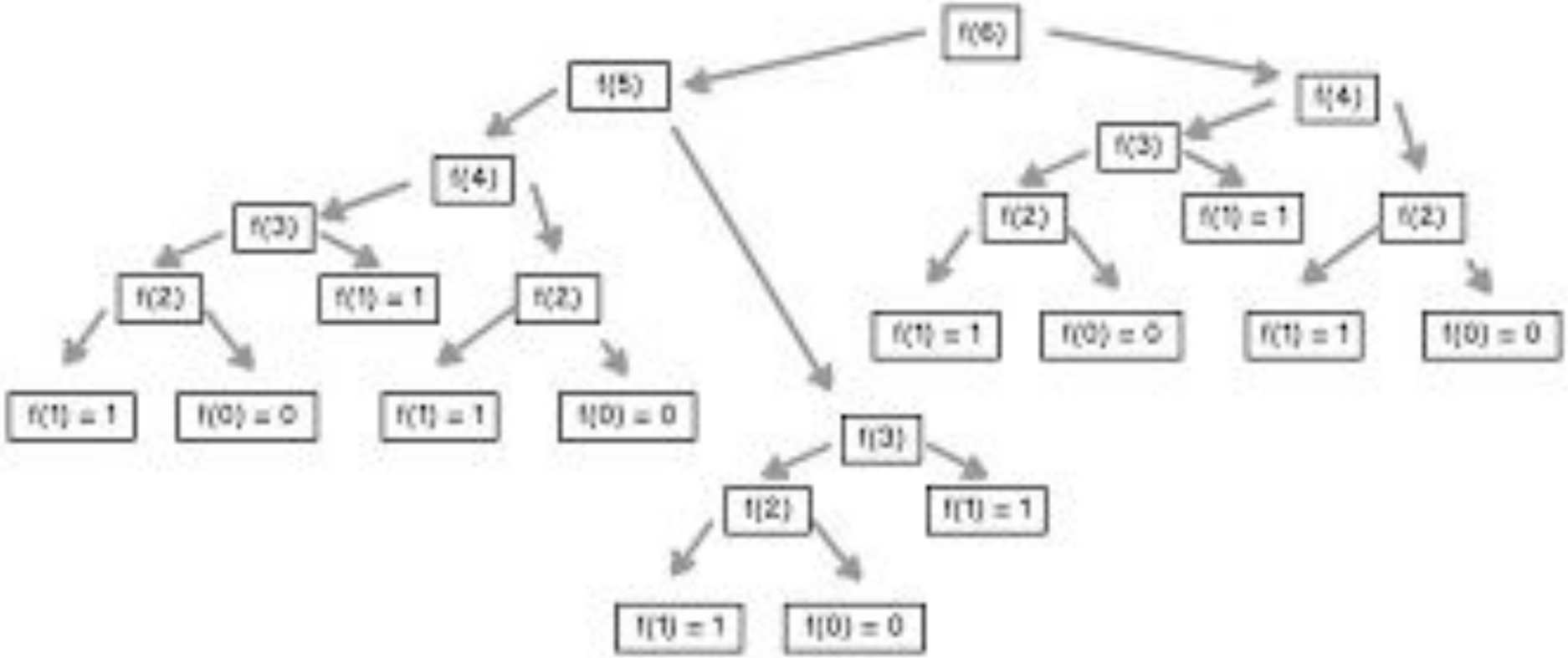
evaluates to 2, which is stored in the variable f, and is returned to main( ), where it is duly printed as Factorial value = 2

- Now perhaps you can see what would happen if the value of a is 3, 4, 5 and so on.
- In case the value of a is 5, main( ) would call rec( ) with 5 as its actual argument, and rec( ) will send back the computed value. But before sending the computed value, rec( ) calls rec( ) and waits for a value to be returned.
- It is possible for the rec( ) that has just been called to call yet another rec( ), the argument x being decreased in value by 1 for each of these recursive calls. We speak of this series of calls to rec( ) as being different invocations of rec( ).
- These successive invocations of the same function are possible because the C compiler keeps track of which invocation calls which. These recursive invocations end finally when the last invocation gets an argument value of 1, which the preceding invocation of rec( ) now uses to calculate its own f value and so on up the ladder.

- Assume that the number entered through scanf( ) is 3. Using Figure let's visualize what exactly happens when the recursive function rec( ) gets called. Go through the figure carefully.
- The first time when rec( ) is called from main( ), x collects 3. From here, since x is not equal to 1, the if block is skipped and rec( ) is called again with the argument ( x – 1 ), i.e. 2. This is a recursive call. Since x is still not equal to 1, rec( ) is called yet another time, with argument (2 - 1). This time as x is 1, control goes back to previous rec( ) with the value 1, and f is evaluated as 2.
- Similarly, each rec( ) evaluates its f from the returned value, and finally 6 is returned to main( ). The sequence would be grasped better by following the arrows shown in Figure 10.1. Let it be clear that while executing the program, there do not exist so many copies of the function rec( ). These have been shown in the figure just to help you keep track of how the control flows during successive recursive calls.




- Recursion may seem strange and complicated at first glance, but it is often the most direct way to code an algorithm, and once you are familiar with recursion, the clearest way of doing so.


f(6)
f(5)
f(4)
f(3)
f(2)
f(1) = 1
f(0) = 0
f(4)
f(3)
f(2)
f(1) = 1
f(2)
f(1) = 1
f(0) = 0
f(1) = 1
f(0) = 0
f(3)
f(2)
f(1) = 1
f(1) = 1
f(0) = 0

# TYPES OF RECURSION

1. Direct Recursion
2. Indirect Recursion
3. Tail Recursion
4. No Tail/ Head Recursion
5. Linear recursion
6. Tree Recursion

# DIRECT RECURSION

- When a function calls itself within the same function repeatedly, it is called the direct recursion.
- **Structure of the direct recursion**

```
fun()
{
// write some code
fun();
// some code
}
```

In the above structure of the direct recursion, the outer fun() function recursively calls the inner fun() function, and this type of recursion is called the direct recursion.

# EXAMPLE

```
#include<stdio.h>
int fibo_num (int i)
{
// if the num i is equal to 0, return 0;
if ( i == 0)
{
return 0;
}
if ( i == 1)
{
return 1;
}
```

```
return fibo_num (i - 1) + fibonacci (i -2);
}
int main ()
{
int i;
// use for loop to get the first 10 fibonacci series
for ( i = 0; i < 10; i++)
{
printf (" %d \t ", fibo_num (i));
}
return 0;
}
```

# INDIRECT RECURSION

When a function is mutually called by another function in a circular manner, the function is called an indirect recursion function.

**Structure of the indirect recursion**

```
fun1()
{
//  write some code
fun2()
}
fun2()
{
// write some code
fun3()
// write some code
}
fun3()
{
// write some code
fun1()
}
```

In this structure, there are four functions, fun1(), fun2(), fun3(). When the fun1() function is executed, it calls the fun2() for its execution. and then, the fun2() function starts its execution calls the fun3() function. In this way, each function leads to another function to makes their execution circularly. And this type of approach is called indirect recursion.

# TAIL RECURSION

- A recursive function is called the tail-recursive if the function makes recursive calling itself, and that recursive call is the last statement executes by the function. After that, there is no function or statement is left to call the recursive function.

```
#include <stdio.h>
// function definition
void fun1( int num)
{
   // if block check the condition
   if (num == 0)
      return;
   else
      printf ("\n Number is: %d", num); // print the number
return fun1 (num - 1);      // recursive call at the end in the fun() function
}
int main ()
{
   fun1(7); // pass 7 as integer argument
   return 0;
}
```

# Non-Tail / Head Recursion

A function is called the non-tail or head recursive if a function makes a recursive call itself, the recursive call will be the first statement in the function. It means there should be no statement or operation is called before the recursive calls. Furthermore, the head recursive does not perform any operation at the time of recursive calling. Instead, all operations are done at the return time.

```
#include <stdio.h>
void head_fun (int num)
{
if ( num > 0 )
{
// Here the head_fun() is the first statement to be called
head_fun (num -1);
printf (" %d", num);
}
}
```

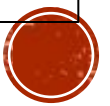
```
int main ()
{
int a = 5;
printf("Use of Non
Tail/Head Recursive function \n");
head_fun (a); // function calling
return 0;
}
```

# Linear recursion

A function is called the linear recursive if the function makes a single call to itself at each time the function runs and grows linearly in proportion to the size of the problem.

```
#include <stdio.h>

#define NUM 7

int rec_num( int *arr, int n)

{

   if (n == 1)

   {

      return arr[0];

      }

   return Max_num (rec_num (arr, n-1), arr[n-1]);

}
```

```
// get the maximum number
int Max_num (int n, int m)
{
   if (n > m)
      return n;
   return m;
}
int main ()
{
   // declare and initialize an array
   int arr[NUM] = { 4, 8, 23, 19, 5, 35, 2};
   int max = rec_num(arr, NUM); // call function
   printf (" The maximum number is: %d\n", max);
 // print the largest number
}
```



# Tree recursion

A function is called the tree recursion, in which the function makes more than one call to itself within the recursive function.

```
#include <stdio.h>

// It is called multiple times insid
e the fibo_num function

int fibo_num (int num)

{

if (num <= 1)

    return num;

return fibo_num (num - 1 ) + fib
o_num(num - 2);

}
```

```
void main()
{
int num = 7;
printf (" Use of Tree Recursion: \n");
// print the number
printf (" The Fibonacci number is: %d", fibo_num(7));
}
```