Functions and Pointers

Introduction

- A function is a self-contained block of statements that perform a coherent task of some kind.
- Every C program can be thought of as a collection of these functions.
- As we noted earlier, using a function is something like hiring a person to do a specific job for you.

```
main( )
{
      message( ) ;
      printf ( "\nCry, and you stop the monotony!" ) ;
}
message( )
{
      printf ( "\nSmile, and the world smiles with you..." ) ;
}
```

And here's the output...

```
Smile, and the world smiles with you...
Cry, and you stop the monotony!
```

Calling Function and Called Function

- Here, main( ) itself is a function and through it we are calling the function message( ).
- 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.

Calling more than one function

```
main( )
{
      printf ( "\nI am in main" ) ;
      italy( ) ;
      brazil( ) ;
      argentina( ) ;
}
italy( )
{
      printf ( "\nI am in italy" ) ;

}
brazil( )
{
      printf ( "\nI am in brazil" ) ;
}
argentina( )
{
      printf ( "\nI am in argentina" ) ;
}
```

The output of the above program when executed would be as under:

```
I am in main
I am in italy
I am in brazil
I am in argentina
```

Important Points

- Any C program contains at least one function.
- If a 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 function calls, the program ends

Important Points

- 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
- Since the compiler always begins the program execution with main( ), every function in a program must be called directly or indirectly by main( ).
- In other words, the main( ) function drives other functions.

```
main( )
{
      printf ( "\nI am in main" ) ;
      italy( ) ;
      printf ( "\nI am finally back in main" ) ;
}
italy( )
{
      printf ( "\nI am in italy" ) ;
      brazil( ) ;
      printf ( "\nI am back in italy" ) ;
}
brazil( )
{
      printf ( "\nI am in brazil" ) ;
      argentina( ) ;
}
argentina( )
{
      printf ( "\nI am in argentina" ) ;
}
```

```
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
```

Summary

- C program is a collection of one or more functions.
- A function gets called when the function name is followed by a semicolon. For example,
- main( )

```
{
    argentina( ) ;
}
```

- A function is defined when function name is followed by a pair of braces in which one or more statements may be present. For example,

```
argentina( )
{
    statement 1 ;
    statement 2 ;
    statement 3 ;
}
```

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

```
main( )
{
    message( ) ;
}
message( )
{
    printf ( "\nCan't imagine life without C" ) ;
    main( ) ;

}
```

- A function can be called any number of times. For example,

```
main( )
{
    message( ) ;
    message( ) ;
}
message( )
{
    printf ( "\nJewel Thief!!" ) ;
}
```

Order of function call vs function definition

- 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,

```
main( )
{
    message1( ) ;
    message2( ) ;
}
message2( )
{
    printf ( "\nBut the butter was bitter" ) ;

 }
 message1( )
 {
     printf ( "\nMary bought some butter" ) ;
 }
```

- A function can call itself. Such a process is called ‘recursion’.
- A function can be called from other 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( ).

```
main( )
{
    printf ( "\nI am in main" ) ;
    argentina( )
    {
        printf ( "\nI am in argentina" ) ;
    }
}
```

Types of Functions

- There are basically two types of functions:
- Library functions Ex. printf( ), scanf( ) etc.
- User-defined functions Ex. argentina( ), brazil( ) etc.
- library functions are nothing but commonly required functions grouped together and stored in what is called a Library
- This library of functions is present on the disk and is written for us by people who write compilers for us.
- Almost always a compiler comes with a library of standard functions.
- The procedure of calling both types of functions is exactly same.

Why Use Functions

- Writing functions avoids rewriting the same code over and over.
- Using functions it becomes easier to write programs and keep track of what they are doing
- Don’t try to cram the entire logic in one function.
- It is a very bad style of programming.
- Instead, break a program into small units and write functions for each of these isolated subdivisions.
- Don’t hesitate to write functions that are called only once.
- What is important is that these functions perform some logically isolated task.

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’

```
/* Sending and receiving values between functions */
main( )
{
      int  a, b, c, sum ;

      printf ( "\nEnter any three numbers " ) ;
      scanf ( "%d %d %d", &a, &b, &c ) ;

      sum = calsum ( a, b, c ) ;



          printf ( "\nSum = %d", sum ) ;
   }

   calsum ( x, y, z )
   int  x, y, z ;
   {
          int  d ;

          d = x + y + z ;
          return ( d ) ;
   }
```

And here is the output...

```
Enter any three numbers 10 20 30
Sum = 60
```

Points to note

- 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

Methods of declaring formal arguments

- There are two methods of declaring the formal arguments. The one that we have used in our program is known as Kernighan and Ritchie (or just K & R) method
- calsum (x, y, z)
- int x, y, z ;
- Another method is,
- calsum (int x, int y, int z)
- This method is called ANSI method and is more commonly used these days.

The Return statement

- 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.
- 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:
- On executing the return statement, it immediately transfers the control back to the calling program
- It returns the value present in the parentheses after return, to the calling program.
- In the above program the value of sum of three numbers is being returned
- 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

```
fun( )
{
    char  ch ;

    printf ( "\nEnter any alphabet " ) ;
    scanf ( "%c", &ch ) ;

    if ( ch >= 65 && ch <= 90 )
        return ( ch ) ;
    else
        return ( ch + 32 ) ;
}
```

- Whenever the control returns from a function some value is definitely returned.
- If a meaningful value is returned then it should be accepted in the calling program by equating the called function to some variable.
- For example,
- sum = calsum (a, b, c) ;
- All the following are valid return statements
- return (a) ;
- return (23) ;
- return (12.34) ;
- 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

- If we want that a called function should not return any value, in that case, we must mention so by using the keyword void as shown below

```
void display( )
{
    printf ( "\nHeads I win..." ) ;
    printf ( "\nTails you lose" ) ;
}
```

- A function can return only one value at a time. Thus, the following statements are invalid.
- return (a, b) ;
- return (x, 12) ;
- We can get around with this using pointer, which will be discussed later

- 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,

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

fun ( int  b )
{
    b = 60 ;

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

The output of the above program would be:

```
60
30
```

Scope Rule of Functions

- In this program it is necessary to pass the value of the variable i to the function display( )
- It will not become automatically available to the function display( )
- 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( )

```
main( )
{
    int  i = 20 ;
    display ( i ) ;
}

display ( int  j )
{
    int  k = 35 ;
    printf ( "\n%d", j ) ;
    printf ( "\n%d", k ) ;
}
```

Scope Rule of Functions

- 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.

Calling Convention

- Calling convention indicates the order in which arguments are passed to a function when a function call is encountered
- There are two possibilities here:
- Arguments might be passed from left to right.
- Arguments might be passed from right to left.
- C language follows the second order.
- 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.

Calling Convention

- However, in some function call the order of passing arguments becomes an important consideration. For example:
- int a = 1 ;
- printf ("%d %d %d", a, ++a, a++) ;
- It appears that this printf( ) would output 1 2 3.
- This however is not the case.
- Surprisingly, it outputs 3 3 1
- This is because C’s calling convention is from right to left

The Header File

- Here we are calling three standard library functions.
- Whenever we call the library functions we must write their prototype before making the call
- This helps the compiler in checking whether the values being passed and returned are as per the prototype declaration
- when the library of functions is provided a set of ‘.h’ files is also provided.
- These files contain the prototypes of library functions

```
#include <conio.h>
clrscr ( ) ;
gotoxy ( 10, 20 ) ;
ch = getch ( a ) ;
```

- On compilation of the above code the compiler reports all errors due to the mismatch between parameters in function call and their corresponding prototypes declared in the file ‘conio.h’.
- You can even open this file and look at the prototypes
- They would appear as shown below:
- void clrscr( ) ;
- void gotoxy (int, int) ;
- int getch( ) ;

- Now consider the following function calls:

```
#include <stdio.h>
int  i = 10, j = 20 ;

printf ( "%d %d %d ", i, j ) ;
printf ( "%d", i, j ) ;
```

- The above functions get successfully compiled even though there is a mismatch in the format specifiers and the variables in the list.
- This is because printf( ) accepts variable number of arguments
- At run-time when the first printf( ) is executed, since there is no variable matching with the last specifier %d, a garbage integer gets printed
- Similarly, in the second printf( ) since the format specifier for j has not been mentioned its value does not get printed.

Advanced Features of Functions

- Function Declaration and Prototypes
- Calling functions by value or by reference
- Recursion

Function Declaration and Prototypes

- Any C function by default returns an int value
- If we desire that a function should return a value other than an int, then it is necessary to explicitly mention so in the calling function as well as in the called function

```
main( )
{
    float  a, b ;

    printf ( "\nEnter any number " ) ;
    scanf ( "%f", &a ) ;

    b = square ( a ) ;
    printf ( "\nSquare of %f is %f", a, b ) ;
}

square ( float  x )
{
    float  y ;

    y = x * x ;
    return ( y ) ;
}
```

And here are three sample runs of this program...

```
Enter any number 3
Square of 3 is 9.000000
Enter any number 1.5
Square of 1.5 is 2.000000
Enter any number 2.5
Square of 2.5 is 6.000000
```

- This happened because any C function, by default, always returns an integer value

```
main( )
{
      float  square ( float ) ;
      float  a, b ;

      printf ( "\nEnter any number " ) ;
      scanf ( "%f", &a ) ;

      b = square ( a ) ;
      printf ( "\nSquare of %f is %f", a, b ) ;
}

float square ( float  x )
{
      float  y ;
      y = x * x ;
      return ( y ) ;
}
```

And here is the output...

```
Enter any number 1.5
Square of 1.5 is 2.250000
Enter any number 2.5
Square of 2.5 is 6.250000
```

Prototype declaration

- Note that the function square( ) must be declared in main( ) as
- float square (float) ;
- This statement is often called the prototype declaration of the square( ) function
- What it means is square( ) is a function that receives a float and returns a float
- We have done the prototype declaration in main( ) because we have called it from main( )
- If it is to be called from multiple different functions then, we would make only one declaration outside all the functions at the beginning of the program
- In some programming situations we want that a called function should not return any value. This is made possible by using the keyword void

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

void gospel( )
{
      printf ( "\nViruses are electronic bandits..." ) ;
      printf ( "\nwho eat nuggets of information..." ) ;
      printf ( "\nand chunks of bytes..." ) ;
      printf ( "\nwhen you least expect..." ) ;
}
```

Call by Value and Call by Reference

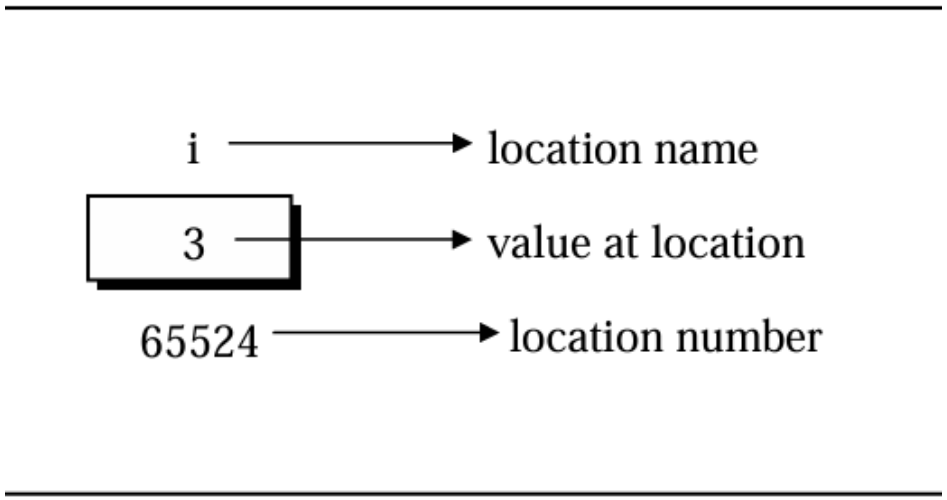
- if you observe carefully, whenever we called a function and passed something to it we have always passed the ‘values’ of variables to the called function
- Such function calls are called ‘calls by value’
- By this what we mean is, on calling a function we are passing values of variables to it
- The examples of call by value are shown below:
- sum = calsum (a, b, c) ;
- f = factr (a) ;

Call by reference

- We have also learnt that variables are stored somewhere in memory.
- So instead of passing the value of a variable, we can also pass the location number (also called address) of the variable to a function
- This feature of C functions needs at least an elementary knowledge of a concept called ‘pointers’

An Introduction to Pointers

- Consider the declaration,
- int i = 3 ;
- This declaration tells the C compiler to:
- Reserve space in memory to hold the integer value.
- Associate the name i with this memory location.
- Store the value 3 at this location.



Address of a number

- The location number 65524 is not a number to be relied upon
- Because some other time the computer may choose a different location for storing the value 3
- We can print this address number through the following program

```
main( )
{
    int  i = 3 ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
```

& and * operators

- ‘&’ used in this statement is C’s ‘address of’ operator
- The expression &i returns the address of the variable i, which in this case happens to be 65524
- Since address can only be non negative, it is printed using %u for unsigned integer
- We have been using the ‘&’ operator all the time in the scanf( ) statement
- The other pointer operator available in C is ‘*’
- called ‘value at address’ operator
- It gives the value stored at a particular address
- also called ‘indirection’ operator

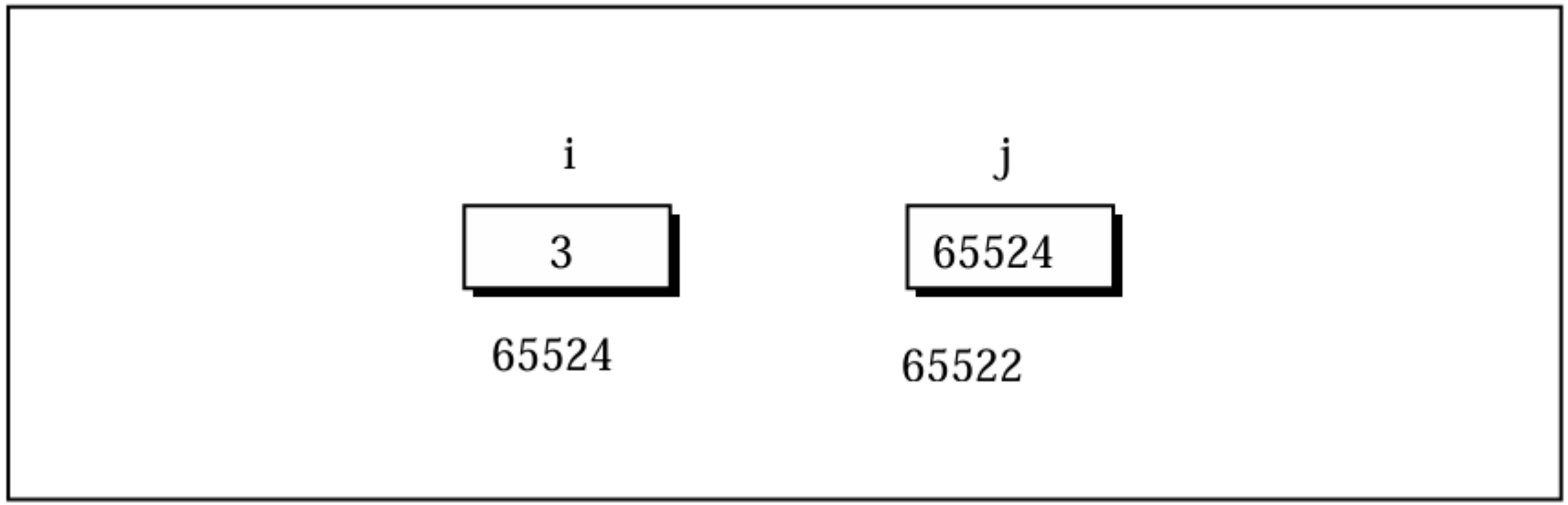
```
main( )
{
      int  i = 3 ;

      printf ( "\nAddress of i = %u", &i ) ;
      printf ( "\nValue of i = %d", i ) ;
      printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
Value of i = 3
```

Pointer Variable

- 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 ;
- 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
- the following memory map 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


i
j
3
65524
65524
65522

Declaration of pointer

- 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
- Thus, int *j would mean, the value at the address contained in j is an int

```
main( )
{
        int  i = 3 ;
        int  *j ;

        j = &i ;
        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nAddress of i = %u", j ) ;
        printf ( "\nAddress of j = %u", &j ) ;
        printf ( "\nValue of j = %u", j ) ;
        printf ( "\nValue of i = %d", i ) ;
        printf ( "\nValue of i = %d", *( &i ) ) ;
        printf ( "\nValue of i = %d", *j ) ;
}
```

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

Value contained by pointers

- Look at the following declarations,
- int *alpha ;
- char *ch ;
- float *s ;
- pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers
- The declaration float *s does not mean that s is going to contain a floating-point value
- What it means is, s is going to contain the address of a floating-point value

Pointer to Pointer

- The concept of pointers can be further extended
- Pointer, we know is a variable that contains address of another variable
- Now this variable itself might be another pointer.
- Thus, we now have a pointer that contains another pointer’s address

```
main( )
{
      int  i = 3, *j, **k ;

      j = &i ;
      k = &j ;
      printf ( "\nAddress of i = %u", &i ) ;
      printf ( "\nAddress of i = %u", j ) ;
      printf ( "\nAddress of i = %u", *k ) ;
      printf ( "\nAddress of j = %u", &j ) ;
      printf ( "\nAddress of j = %u", k ) ;
      printf ( "\nAddress of k = %u", &k ) ;
      printf ( "\nValue of j  = %u", j ) ;
      printf ( "\nValue of k  = %u", k ) ;
      printf ( "\nValue of i  = %d", i ) ;
      printf ( "\nValue of i  = %d", * ( &i ) ) ;
      printf ( "\nValue of i  = %d", *j ) ;
      printf ( "\nValue of i  = %d", **k ) ;
}
```

```
Address of i = 65524

Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j  = 65524
Value of k  = 65522
Value of i  = 3
Value of i  = 3
Value of i  = 3
Value of i  = 3
```

Important Points

- Here, i is an ordinary int, j is a pointer to an int (often called an integer pointer), whereas k is a pointer to an integer pointer
- We can extend the above program still further by creating a pointer to a pointer to an integer pointer
- There is no limit on how far can we go on extending this definition
- the point up to which one can comprehend is usually a pointer to a pointer
- Beyond this one rarely requires to extend the definition of a pointer

Back to Function Calls

- get back to what we had originally set out to learn—the two types of function calls—
- call by value and
- call by reference
- Arguments can generally be passed to functions in one of the two ways:
- sending the values of the arguments
- sending the addresses of the arguments

Call by Value

- In the first method the ‘value’ of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function
- With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function
- The following program illustrates the ‘Call by Value’

```
main( )
{
      int  a = 10, b = 20 ;

      swapv ( a, b ) ;
      printf ( "\na = %d b = %d", a, b ) ;
}

swapv ( int  x, int  y )
{
      int  t ;

      t = x ;
      x = y ;
      y = t ;

      printf ( "\nx = %d y = %d", x, y ) ;
}
```

```
x = 20 y = 10
a = 10 b = 20
```

call by reference

- In the second method (call by reference) 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
- Note that this program manages to exchange the values of a and b using their addresses stored in x and y

```
main( )
{
      int  a = 10, b = 20 ;

      swapr ( &a, &b ) ;
      printf ( "\na = %d b = %d", a, b ) ;
}

swapr( int  *x, int  *y )
{
      int  t ;

      t = *x ;
      *x = *y ;
      *y = t ;
}
```

The output of the above program would be:

a = 20 b = 10

Returning more than one value

- Usually in C programming we make a call by value. This means that in general you cannot alter the actual arguments.
- But if desired, it can always be achieved through a call by reference.
- Using a call by reference intelligently we can make a function return more than one value at a time

```
main( )
{
      int  radius ;
      float  area, perimeter ;

      printf ( "\nEnter radius of a circle " ) ;
      scanf ( "%d", &radius ) ;
      areaperi ( radius, &area, &perimeter ) ;

      printf ( "Area = %f", area ) ;
      printf ( "\nPerimeter = %f", perimeter ) ;
}

areaperi ( int  r, float  *a, float  *p )
{
      *a = 3.14 * r * r ;
      *p = 2 * 3.14 * r ;
}
```

And here is the output...

```
Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000
```

Conclusions

- If we want that the value of an actual argument should not get changed in the function being called, pass the actual argument by value
- If we want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference
- If a function is to be made to return more than one value at a time then return these values indirectly by using a call by reference

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’
- For example, factorial of a number
- 4 factorial is 4 * 3 * 2 * 1
- also 4! = 4 * 3!
- Thus factorial of a number can be expressed in the form of itself

```
main( )
{
      int  a, fact ;

      printf ( "\nEnter any number " ) ;
      scanf ( "%d", &a ) ;

      fact = factorial ( a ) ;
      printf ( "Factorial value = %d", fact ) ;
}
```

```
factorial ( int  x )
{
      int  f = 1, i ;

      for ( i = x ; i >= 1 ; i-- )
            f = f * i ;

      return ( f ) ;
}
```

And here is the output...

```
Enter any number 3
Factorial value = 6
```

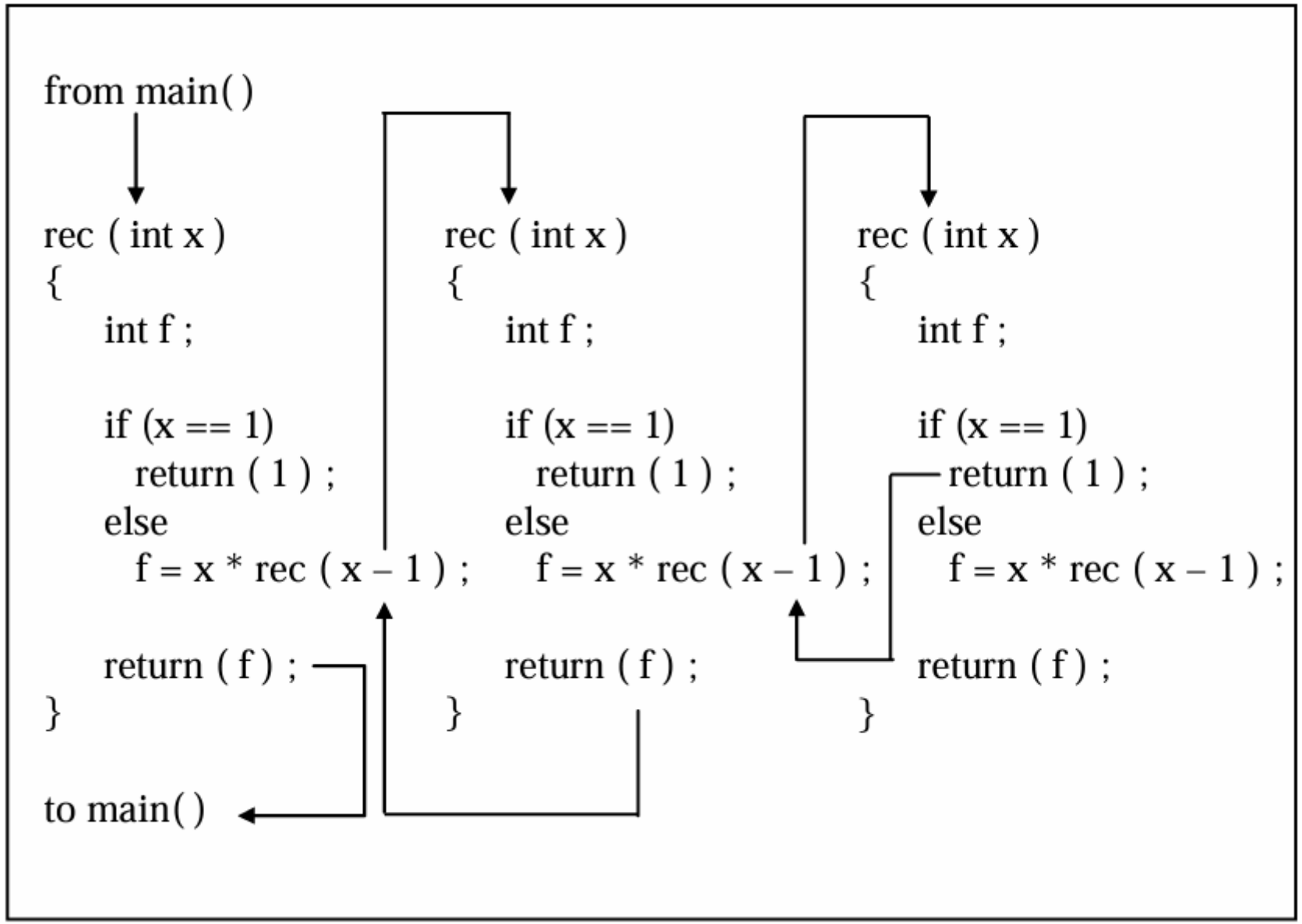
```
main( )
{
      int  a, fact ;

      printf ( "\nEnter any number " ) ;
      scanf ( "%d", &a ) ;

      fact = rec ( a ) ;
      printf ( "Factorial value = %d", fact ) ;
}
```

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

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

      return ( f ) ;
}
```

And here is the output for four runs of the program

```
Enter any number 1
Factorial value = 1
Enter any number 2
Factorial value = 2
Enter any number 3
Factorial value = 6
Enter any number 5
Factorial value = 120
```

rec (5) returns (5 times rec (4),
which returns (4 times rec (3),
which returns (3 times rec (2),
which returns (2 times rec (1),
which returns (1)))))


from main( )
rec (int x)
{
int f ;
if (x == 1)
return (1) ;
else
f = x * rec (x – 1) ;
return (f) ;
}
to main( )
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) ;
}

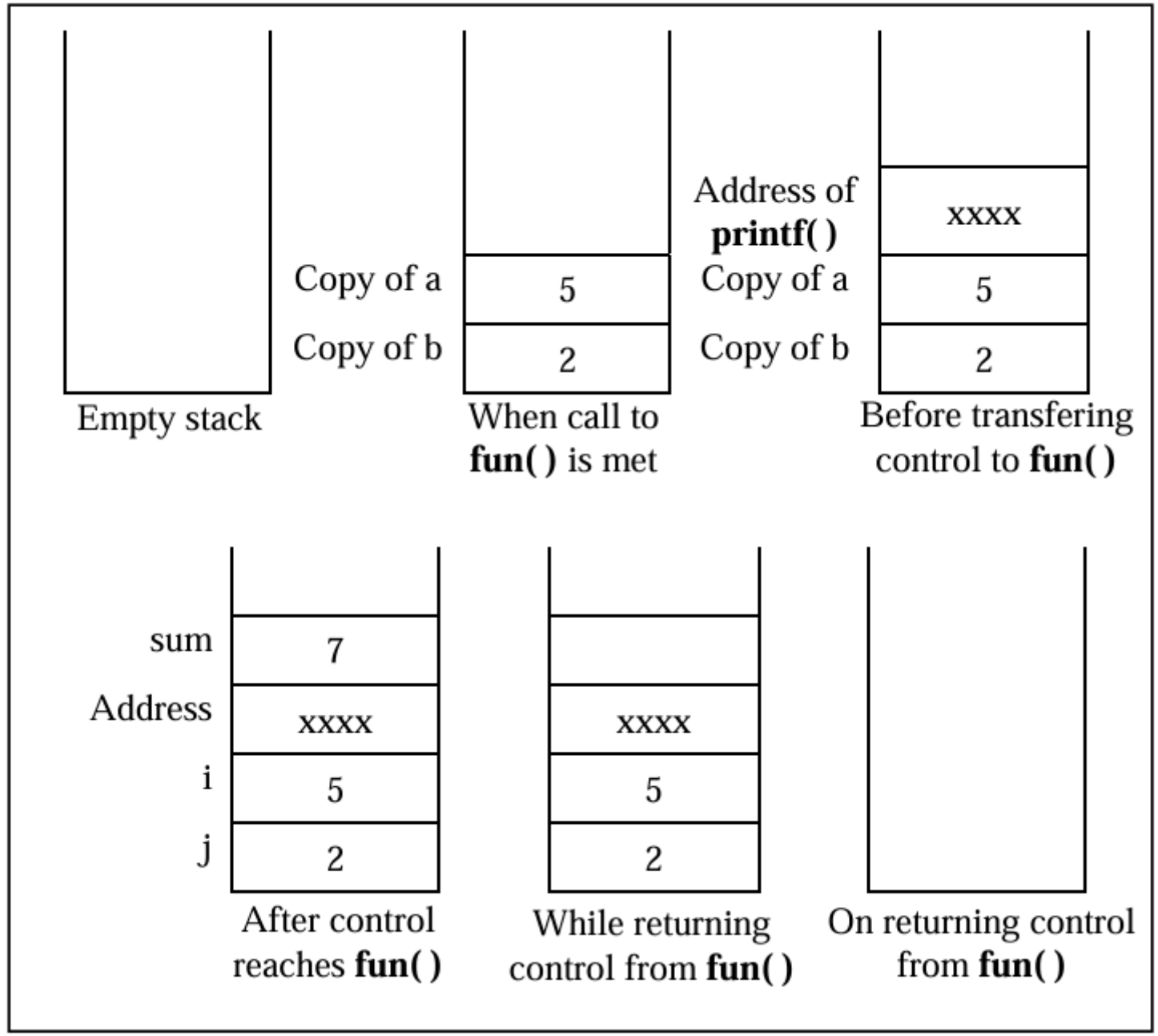
More about recursion

- 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

Recursion and Stack

- There are different ways in which data can be organized
- All these different ways of organizing the data are known as data structures.
- The compiler uses one such data structure called stack for implementing normal as well as recursive function calls
- A stack is a Last In First Out (LIFO) data structure.
- This means that the last item to get stored on the stack (often called Push operation) is the first one to get out of it (often called as Pop operation)
- You can compare this to the stack of plates in a cafeteria—

```
main( )
{
        int a = 5, b = 2, c ;
        c = add ( a, b ) ;
        printf ( "sum = %d", c ) ;
}
add ( int i, int j )
{
        int sum ;
        sum = i + j ;
        return sum ;
}
```


Copy of a
Copy of b
5
2
Empty stack
When call to
fun( ) is met
Address of
printf( )
Copy of a
Copy of b
xxxx
5
2
Before transfering
control to fun( )
sum
Address
i
j
7
xxxx
5
2
After control
reaches fun( )
xxxx
5
2
While returning
control from fun( )
On returning control
from fun( )

Recursive Calls

- The recursive calls are no different.
- Whenever we make a recursive call the parameters and the return address gets pushed on the stack.
- The stack gets unwound when the control returns from the called function.
- Thus during every recursive function call we are working with a fresh set of parameters.
- Also, note that while writing recursive functions you must have an if statement somewhere in the recursive function to force the function to return without recursive call being executed

if statement in recursive call

- Also, note that while writing recursive functions you must have an if statement somewhere in the recursive function to force the function to return without recursive call being executed
- If you don’t do this and you call the function, you will fall in an indefinite loop
- and the stack will keep on getting filled with parameters and the return address each time there is a call
- Soon the stack would become full and you would get a run-time error indicating that the stack has become full
- This is a very common error while writing recursive functions