5 Functions & Pointers

- What is a Function
  - Why Use Functions
- Passing Values between Functions
- Scope Rule of Functions
- Calling Convention
- One Dicey Issue
- Advanced Features of Functions
  - Function Declaration and Prototypes
  - Call by Value and Call by Reference
  - An Introduction to Pointers
  - Pointer Notation
  - Back to Function Calls
  - Conclusions
  - Recursion
- Adding Functions to the Library
- Summary
- Exercise

Knowingly or unknowingly we rely on so many persons for so many things. Man is an intelligent species, but still cannot perform all of life's tasks all alone. He has to rely on others. You may call a mechanic to fix up your bike, hire a gardener to mow your lawn, or rely on a store to supply you groceries every month. A computer program (except for the simplest one) finds itself in a similar situation. It cannot handle all the tasks by itself. Instead, it requests other program like entities—called 'functions' in C—to get its tasks done. In this chapter we will study these functions. We will look at a variety of features of these functions, starting with the simplest one and then working towards those that demonstrate the power of C functions.

What is a Function

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. Sometimes the interaction with this person is very simple; sometimes it's complex.

Suppose you have a task that is always performed exactly in the same way—say a bimonthly servicing of your motorbike. When you want it to be done, you go to the service station and say, "It's time, do it now". You don't need to give instructions, because the mechanic knows his job. You don't need to be told when the job is done. You assume the bike would be serviced in the usual way, the mechanic does it and that's that.

Let us now look at a simple C function that operates in much the same way as the mechanic. Actually, we will be looking at two things—a function that calls or activates the function and the function itself.

```
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!
```

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.

If you have grasped the concept of 'calling' a function you are prepared for a call to more than one function. Consider the following example:

```
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
```

From this program a number of conclusions can be drawn:

- 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.

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, if you understand what I mean. No? Well, let's illustrate with an example.

```
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" ) ;
}
```

And the output would look like...

```
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
```

Here, **main( )** calls other functions, which in turn call still other functions. Trace carefully the way control passes from one function to another. 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.

Let us now summarize what we have learnt so far.

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

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

(c) 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 ;
}
```

(d) 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( ) ;

}
```

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

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

(f) 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" ) ;
}
```

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.

(g) A function can call itself. Such a process is called 'recursion'. We would discuss this aspect of C functions later in this chapter.

(h) 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" ) ;
    }
}
```

(i) 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

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

Why write separate functions at all? Why not squeeze the entire logic into one function, **main( )**? Two reasons:

(a) Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If later in the program you want to calculate the area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.

(b) 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. Separating the code into modular functions also makes the program easier to design and understand.

What is the moral of the story? 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 functions that we have used so far haven't been very flexible. We call them and they do what they are designed to do. Like our mechanic who always services the motorbike in exactly the same way, we haven't been able to influence the functions in the way they carry out their tasks. It would be nice to have a little more control over what functions do, in the same way it would be nice to be able to tell the mechanic, "Also change the engine oil, I am going for an outing". In short, now we want to communicate between the 'calling' and the 'called' 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'.

Consider the following program. In this program, in **main( )** we receive the values of **a**, **b** and **c** through the keyboard and then output the sum of **a**, **b** and **c**. However, the calculation of sum is done in a different function called **calsum( )**. If sum is to be calculated in **calsum( )** and values of **a**, **b** and **c** are received in **main( )**, then we must pass on these values to **calsum( )**, and once **calsum( )** calculates the sum we must return it from **calsum( )** back to **main( )**.

```
/* 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
```

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**:

```
calsum ( x, y, z )
int  x, y, 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.

(c) 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.

(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 program.
(2) It returns the value present in the parentheses after **return**, to th3e calling program. In the above program the value of sum of three numbers is being returned.

(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.

```
fun( )
{
    char  ch ;

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

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

In this function different **return** statements will be executed depending on whether **ch** is capital or not.

(f) 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 ) ;
```

(g) 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.

(h) 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" ) ;
}
```

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

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

There is a way to get around this limitation, which would be discussed later in this chapter when we learn pointers.

(j) 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
```

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

Look at the following program

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

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

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.

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:

(a) Arguments might be passed from left to right.
(b) 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. 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. That is, 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.

One Dicey Issue

Consider the following function calls:

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

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. But since we don't define the library functions (we merely call them) we may not know the prototypes of library functions. Hence when the library of functions is provided a set of '.h' files is also provided. These files contain the prototypes of library functions. But why multiple files? Because the library functions are divided into different groups and one file is provided for each group. For example, prototypes of all input/output functions are provided in the file 'stdio.h', prototypes of all mathematical functions are provided in the file 'math.h', etc.

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 (sometimes 2 arguments, sometimes 3 arguments, etc.), and even with the mismatch above the call still matches with the prototype of **printf( )** present in 'stdio.h'. 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

With a sound basis of the preliminaries of C functions, let us now get into their intricacies. Following advanced topics would be considered here.

(a) Function Declaration and Prototypes
(b) Calling functions by value or by reference
(c) Recursion

Let us understand these features one by one.

Function Declaration and Prototypes

Any C function by default returns an **int** value. More specifically, whenever a call is made to a function, the compiler assumes that this function would return a value of the type **int**. 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. Suppose we want to find out square of a number using a function. This is how this simple program would look like:

```
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
```

The first of these answers is correct. But square of 1.5 is definitely not 2. Neither is 6 a square of 2.5. This happened because any C function, by default, always returns an integer value. Therefore, even though the function **square( )** calculates the square of 1.5 as 2.25, the problem crops up when this 2.25 is to be returned to **main( )**. **square( )** is not capable of returning a **float** value. How do we overcome this? The following program segment illustrates how to make **square( )** capable of returning a **float** 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
```

Now the expected answers i.e. 2.25 and 6.25 are obtained. 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( )**. There is a possibility that we may call **square( )** from several other functions other than **main( )**. Does this mean that we would need prototype declaration of **square( )** in all these functions. No, in such a case we would make only one declaration outside all the functions at the beginning of the program.

In practice you may seldom be required to return a value other than an **int**, but just in case you are required to, employ the above method. In some programming situations we want that a called function should not return any value. This is made possible by using the keyword **void**. This is illustrated in the following program.

```
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..." ) ;
}
```

Here, the **gospel( )** function has been defined to return **void**; means it would return nothing. Therefore, it would just flash the four messages about viruses and return the control back to the **main( )** function.

Call by Value and Call by Reference

By now we are well familiar with how to call functions. But, 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 ) ;
```

We have also learnt that variables are stored somewhere in memory. So instead of passing the value of a variable, can we not pass the location number (also called address) of the variable to a function? If we were able to do so it would become a 'call by reference'. What purpose a 'call by reference' serves we would find out a little later. First we must equip ourselves with knowledge of how to make a 'call by reference'. This feature of C functions needs at least an elementary knowledge of a concept called 'pointers'. So let us first acquire the basics of pointers after which we would take up this topic once again.

An Introduction to Pointers

Which feature of C do beginners find most difficult to understand? The answer is easy: pointers. Other languages have pointers but few use them so frequently as C does. And why not? It is C's clever use of pointers that makes it the excellent language it is.

The difficulty beginners have with pointers has much to do with C's pointer terminology than the actual concept. For instance, when a C programmer says that a certain variable is a "pointer", what does that mean? It is hard to see how a variable can point to something, or in a certain direction.

It is hard to get a grip on pointers just by listening to programmer's jargon. In our discussion of C pointers, therefore, we will try to avoid this difficulty by explaining pointers in terms of programming concepts we already understand. The first thing we want to do is explain the rationale of C's pointer notation.

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.

We may represent **i**'s location in memory by the following memory map.

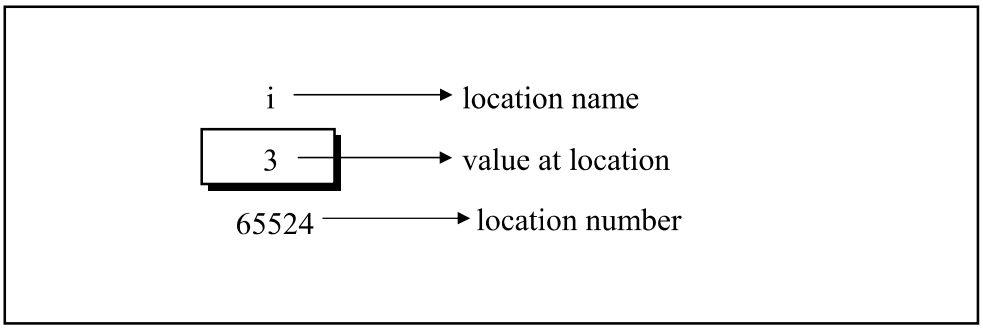

Figure 5.1

We see that the computer has selected memory location 65524 as the place to store the value 3. 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. The important point is, **i**'s address in memory is a number.

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
```

Look at the first **printf( )** statement carefully. '&' 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 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an 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. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program:

```
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
```

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 following memory map would illustrate the contents of **i** and **j**.

Figure 5.2

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**.

Here is a program that demonstrates the relationships we have been discussing.

```
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 ) ;
}
```

The output of the above program would be:

```
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
```

Work through the above program carefully, taking help of the memory locations of **i** and **j** shown earlier. This program summarizes everything that we have discussed so far. If you don't understand the program's output, or the meanings of **&i**, **&j**, ***j** and ***(&i)**, re-read the last few pages. Everything we say about C pointers from here onwards will depend on your understanding these expressions thoroughly.

Look at the following declarations,

```
int  *alpha ;
char  *ch ;
float  *s ;
```

Here, **alpha, ch** and **s** are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers. Now we can put these two facts together and say—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. Similarly, **char *ch** means that **ch** is going to contain the address of a char value. Or in other words, the value at address stored in **ch** is going to be a **char**.

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. The following example should make this point clear.

```
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 ) ;
}
```

The output of the above program would be:

```
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
```

Figure 5.3 would help you in tracing out how the program prints the above output.

Remember that when you run this program the addresses that get printed might turn out to be something different than the ones shown in the figure. However, with these addresses too the relationship between **i**, **j** and **k** can be easily established.

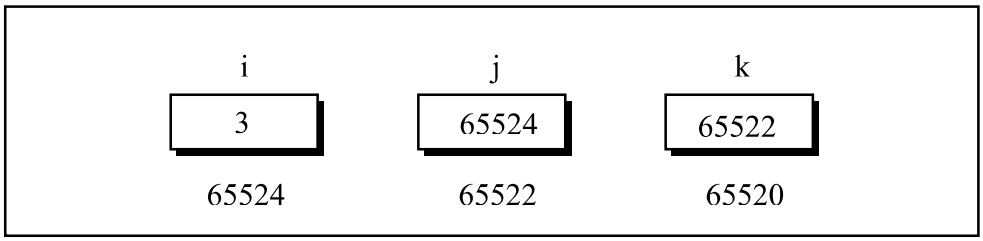

Figure 5.3

Observe how the variables **j** and **k** have been declared,

```
int  i, *j, **k ;
```

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. In principle, you would agree that likewise there could exist a pointer to a pointer to a pointer to a pointer to a pointer. There is no limit on how far can we go on extending this definition. Possibly, till the point we can comprehend it. And that point of comprehension is usually a pointer to a pointer. Beyond this one rarely requires to extend the definition of a pointer. But just in case...

Back to Function Calls

Having had the first tryst with pointers let us now 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:

(a) sending the values of the arguments
(b) sending the addresses of the arguments

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 ) ;
}
```

The output of the above program would be:

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

Note that values of **a** and **b** remain unchanged even after exchanging the values of **x** and **y**.

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. The following program illustrates this fact.

```
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
```

Note that this program manages to exchange the values of **a** and **b** using their addresses stored in **x** and **y**.

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, which is not possible ordinarily. This is shown in the program given below.

```
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
```

Here, we are making a mixed call, in the sense, we are passing the value of **radius** but, addresses of **area** and **perimeter**. And since we are passing the addresses, any change that we make in values stored at addresses contained in the variables **a** and **p**, would make

the change effective in **main( )**. That is why when the control returns from the function **areaperi( )** we are able to output the values of **area** and **perimeter**.

Thus, we have been able to indirectly return two values from a called function, and hence, have overcome the limitation of the **return** statement, which can return only one value from a function at a time.

Conclusions

From the programs that we discussed here we can draw the following conclusions:

(a) 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.

(b) If we want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference.

(c) 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', recursion is thus the process of defining something in terms of itself.

Let us now see a simple example of recursion. Suppose we want to calculate the factorial value of an integer. As we know, the

factorial of a number is the product of all the integers between 1 and that number. For example, 4 factorial is 4 * 3 * 2 * 1. This can also be expressed as 4! = 4 * 3! where '!' stands for factorial. Thus factorial of a number can be expressed in the form of itself. Hence this can be programmed using recursion. However, before we try to write a recursive function for calculating factorial let us take a look at the non-recursive function for calculating the factorial value of an integer.

```
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
```

Work through the above program carefully, till you understand the logic of the program properly. Recursive factorial function can be understood only if you are thorough with the above logic.

Following is the recursive version of the function to calculate the factorial value.

```
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
```

Let us understand this recursive factorial function thoroughly. 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. So we might say what happens is,

```
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 ) ) ) ) )
```

Foxed? Well, that is recursion for you in its simplest garbs. I hope you agree that it's difficult to visualize how the control flows from one function call to another. Possibly Figure 5.4 would make things a bit clearer.

Assume that the number entered through **scanf( )** is 3. Using Figure 5.4 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 5.4. 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.

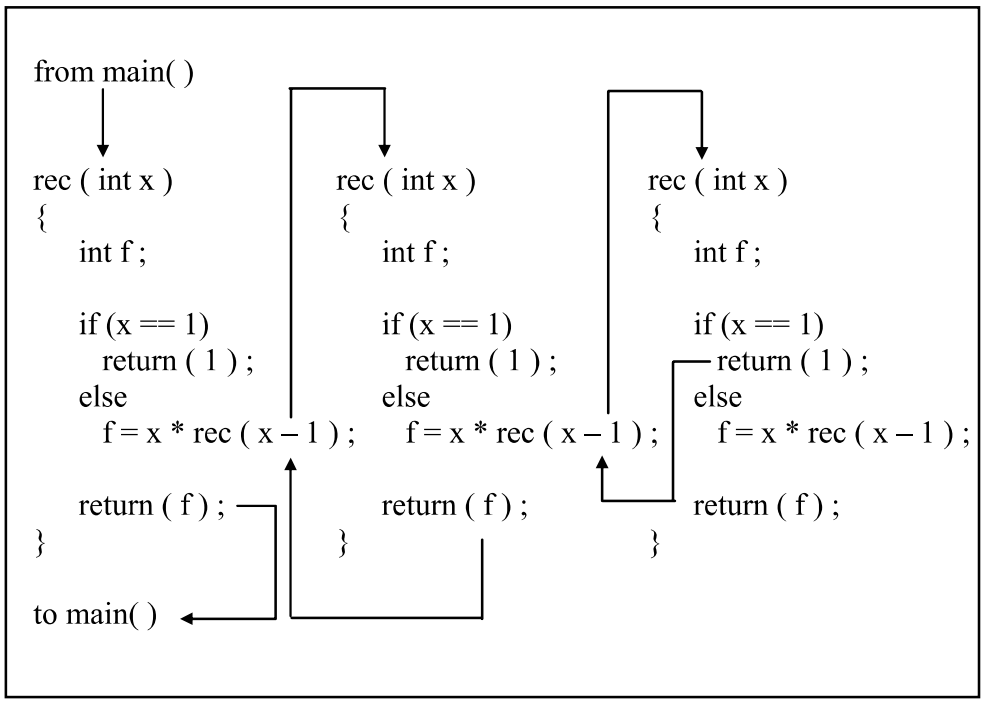

Figure 5.4

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. For example, if you are to store five numbers then we can store them in five different variables, an array, a linked list, a binary tree, etc. 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—the last plate that goes on the stack is the first one to get out of it. Now let us see how the stack works in case of the following program.

```
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 ;
}
```

In this program before transferring the execution control to the function **fun( )** the values of parameters **a** and **b** are pushed onto the stack. Following this the address of the statement **printf( )** is pushed on the stack and the control is transferred to **fun( )**. It is necessary to push this address on the stack. In **fun( )** the values of **a** and **b** that were pushed on the stack are referred as **i** and **j**. In **fun( )** the local variable **sum** gets pushed on the stack. When value of **sum** is returned **sum** is popped up from the stack. Next the address of the statement where the control should be returned is popped up from the stack. Using this address the control returns to the **printf( )** statement in **main( )**. Before execution of **printf( )** begins the two integers that were earlier pushed on the stack are now popped off.

How the values are being pushed and popped even though we didn't write any code to do so? Simple—the compiler on

encountering the function call would generate code to push parameters and the address. Similarly, it would generate code to clear the stack when the control returns back from **fun( )**. Figure 5.5 shows the contents of the stack at different stages of execution.

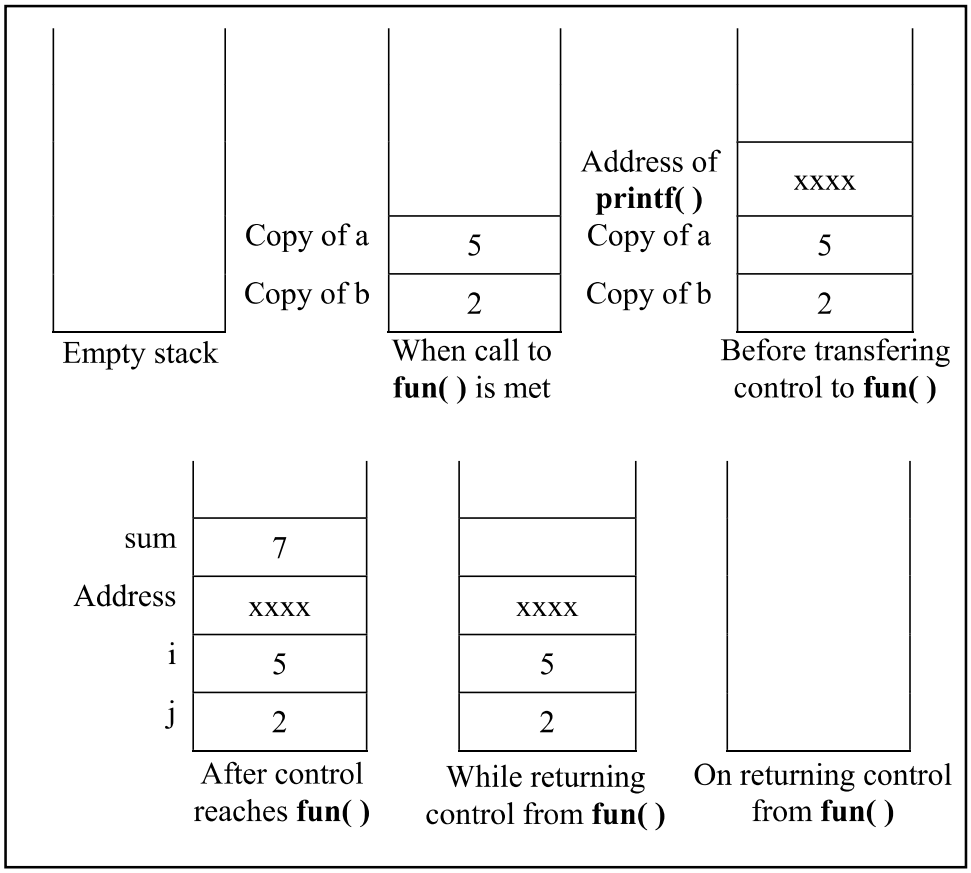

Figure 5.5

Note that in this program popping of **sum** and address is done by **fun( )**, whereas popping of the two integers is done by **main( )**. When it is done this way it is known as 'CDecl Calling Convention'. There are other calling conventions as well where instead of **main( )**, **fun( )** itself clears the two integers. The calling convention also decides whether the parameters being passed to the function are pushed on the stack in left-to-right or right-to-left order. The standard calling convention always uses the right-to-left

order. Thus during the call to **fun( )** firstly value of **b** is pushed to the stack, followed by the value of **a**.

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 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. My advice is to use **printf( )** statement liberally during the development of recursive function, so that you can watch what is going on and can abort execution if you see that you have made a mistake.

Adding Functions to the Library

Most of the times we either use the functions present in the standard library or we define our own functions and use them. Can we not add our functions to the standard library? And would it make any sense in doing so? We can add user-defined functions to the library. It makes sense in doing so as the functions that are to be added to the library are first compiled and then added. When we use these functions (by calling them) we save on their compilation time as they are available in the library in the compiled form.

Let us now see how to add user-defined functions to the library. Different compilers provide different utilities to add/delete/modify functions in the standard library. For example, Turbo C/C++

compilers provide a utility called 'tlib.exe' (Turbo Librarian). Let us use this utility to add a function **factorial( )** to the library.

Given below are the steps to do so:

(a) Write the function definition of **factorial( )** in some file, say 'fact.c'.

```
int factorial ( int num )
{
    int  i, f = 1 ;
    for ( i = 1 ; i <= num ; i++ )
          f = f * i ;
    return ( f ) ;
}
```

(b) Compile the 'fact.c' file using Alt F9. A new file called 'fact.obj' would get created containing the compiled code in machine language.

(c) Add the function to the library by issuing the command

```
C:\>tlib  math.lib + c:\fact.obj
```

Here, 'math.lib' is a library filename, + is a switch, which means we want to add new function to library and 'c:\fact.obj' is the path of the '.obj' file.

(d) Declare the prototype of the **factorial( )** function in the header file, say 'fact.h'. This file should be included while calling the function.

(e) To use the function present inside the library, create a program as shown below:

```
#include "c:\\fact.h"
main( )
```

```
{
    int f ;
    f = factorial ( 5 ) ;
    printf ( "%d", f ) ;
}
```

(f) Compile and execute the program using Ctrl F9.

If we wish we can delete the existing functions present in the library using the minus (-) switch.

Instead of modifying the existing libraries we can create our own library. Let's see how to do this. Let us assume that we wish to create a library containing the functions **factorial( )**, **prime( )** and **fibonacci( )**. As their names suggest, **factorial( )** calculates and returns the factorial value of the integer passed to it, **prime( )** reports whether the number passed to it is a prime number or not and **fibonacci( )** prints the first **n** terms of the Fibonacci series, where **n** is the number passed to it. Here are the steps that need to be carried out to create this library. Note that these steps are specific to Turbo C/C++ compiler and would vary for other compilers.

(a) Define the functions **factorial( )**, **prime( )** and **fibonacci( )** in a file, say 'myfuncs.c'. Do not define **main( )** in this file.

(b) Create a file 'myfuncs.h' and declare the prototypes of **factorial( )**, **prime( )** and **fibonacci( )** in it as shown below:

```
int factorial ( int ) ;
int prime ( int ) ;
void fibonacci ( int ) ;
```

(c) From the Options menu select the menu-item 'Application'. From the dialog that pops us select the option 'Library'. Select OK.

(d) Compile the program using Alt F9. This would create the library file called 'myfuncs.lib'.

That's it. The library now stands created. Now we have to use the functions defined in this library. Here is how it can be done.

(a) Create a file, say 'sample.c' and type the following code in it.

```
#include "myfuncs.h"
main( )
{
    int  f, result ;
    f = factorial ( 5 ) ;
    result = prime ( 13 ) ;
    fibonacci ( 6 ) ;
    printf ( "\n%d %d", f, result ) ;
}
```

Note that the file 'myfuncs.h' should be in the same directory as the file 'sample.c'. If not, then while including 'myfuncs.h' mention the appropriate path.

(b) Go to the 'Project' menu and select 'Open Project…' option. On doing so a dialog would pop up. Give the name of the project, say 'sample.prj' and select OK.

(c) From the 'Project' menu select 'Add Item'. On doing so a file dialog would appear. Select the file 'sample.c' and then select 'Add'. Also add the file 'myfuncs.lib' in the same manner. Finally select 'Done'.

(d) Compile and execute the project using Ctrl F9.

Summary

(a) To avoid repetition of code and bulky programs functionally related statements are isolated into a function.
(b) Function declaration specifies what is the return type of the function and the types of parameters it accepts.
(c) Function definition defines the body of the function.
(d) Variables declared in a function are not available to other functions in a program. So, there won't be any clash even if we give same name to the variables declared in different functions.
(e) Pointers are variables which hold addresses of other variables.
(f) A function can be called either by value or by reference.
(g) Pointers can be used to make a function return more than one value simultaneously.
(h) Recursion is difficult to understand, but in some cases offer a better solution than loops.
(i) Adding too many functions and calling them frequently may slow down the program execution.

Exercise

Simple functions, Passing values between functions

[A] What would be the output of the following programs:

(a)
```
main( )
{
    printf ( "\nOnly stupids use C?" ) ;
    display( ) ;
}
display( )
{
    printf ( "\nFools too use C!" ) ;
    main( ) ;
}
```

(b)
```
main( )
{
    printf ( "\nC to it that C survives" ) ;
    main( ) ;
}
```

(c)
```
main( )
{
    int  i = 45, c ;
    c = check ( i ) ;
    printf ( "\n%d", c ) ;
}
check ( int  ch )
{
    if ( ch >= 45 )
        return ( 100 ) ;
    else
        return ( 10 * 10 ) ;
}
```

(d)
```
main( )
{
    int  i = 45, c ;
    c = multiply ( i * 1000 ) ;
    printf ( "\n%d", c ) ;
}
check ( int  ch )
{
    if ( ch >= 40000 )
        return ( ch / 10 ) ;
    else
        return ( 10 ) ;
}
```

[B] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
```

```
        int  i = 3, j = 4, k, l ;
        k = addmult ( i, j ) ;
        l = addmult ( i, j ) ;
        printf ( "\n%d %d", k, l ) ;
    }
    addmult ( int  ii, int  jj )
    {
        int  kk, ll ;
        kk = ii + jj ;
        ll = ii * jj ;
        return ( kk, ll ) ;
    }
```

(b)
```
    main( )
    {
        int  a ;
        a = message( ) ;
    }
    message( )
    {
        printf ( "\nViruses are written in C" ) ;
        return ;
    }
```

(c)
```
    main( )
    {
        float  a = 15.5 ;
        char  ch = 'C' ;
        printit ( a, ch ) ;
    }
    printit ( a, ch )
    {
        printf ( "\n%f %c", a, ch ) ;
    }
```

(d)
```
    main( )
    {
        message( ) ;
```

```
        message( ) ;
    }
    message( ) ;
    {
        printf ( "\nPraise worthy and C worthy are synonyms" ) ;
    }
```

(e)
```
    main( )
    {
        let_us_c( )
        {
            printf ( "\nC is a Cimple minded language !" ) ;
            printf ( "\nOthers are of course no match !" ) ;
        }
    }
```

(f)
```
    main( )
    {
        message( message ( ) ) ;
    }
    void message( )
    {
        printf ( "\nPraise worthy and C worthy are synonyms" ) ;
    }
```

[C] Answer the following:

(a) Is this a correctly written function:

```
sqr ( a ) ;
int  a ;
{
    return ( a * a ) ;
}
```

(b) State whether the following statements are True or False:

1. The variables commonly used in C functions are available to all the functions in a program.

2. To return the control back to the calling function we must use the keyword **return**.

3. The same variable names can be used in different functions without any conflict.

4. Every called function must contain a **return** statement.

5. A function may contain more than one **return** statements.

6. Each **return** statement in a function may return a different value.

7. A function can still be useful even if you don't pass any arguments to it and the function doesn't return any value back.

8. Same names can be used for different functions without any conflict.

9. A function may be called more than once from any other function.

10. It is necessary for a function to return some value.

[D] Answer the following:

(a) Write a function to calculate the factorial value of any integer entered through the keyboard.

(b) Write a function **power (a, b)**, to calculate the value of **a** raised to **b**.

(c) Write a general-purpose function to convert any given year into its roman equivalent. The following table shows the roman equivalents of decimal numbers:

Decimal	Roman	Decimal	Roman
1	i	100	c
5	v	500	d
10	x	1000	m
50	l		

Example:

Roman equivalent of 1988 is mdcccclxxxviii
Roman equivalent of 1525 is mdxxv

(d) Any year is entered through the keyboard. Write a function to determine whether the year is a leap year or not.

(e) A positive integer is entered through the keyboard. Write a function to obtain the prime factors of this number.

For example, prime factors of 24 are 2, 2, 2 and 3, whereas prime factors of 35 are 5 and 7.

Function Prototypes, Call by Value/Reference, Pointers

[E] What would be the output of the following programs:

(a)
```
main( )
{
    float  area ;
    int  radius = 1 ;
    area = circle ( radius ) ;
    printf ( "\n%f", area ) ;
}
circle ( int  r )
```

```
    {
        float  a ;
        a = 3.14 * r * r ;
        return ( a ) ;
    }
```

(b)
```
main( )
{
    void  slogan( ) ;
    int  c = 5 ;
    c = slogan( ) ;
    printf ( "\n%d", c ) ;
}
void slogan( )
{
    printf ( "\nOnly He men use C!" ) ;
}
```

[F] Answer the following:

(a) Write a function which receives a **float** and an **int** from **main( )**, finds the product of these two and returns the product which is printed through **main( )**.

(b) Write a function that receives 5 integers and returns the sum, average and standard deviation of these numbers. Call this function from **main( )** and print the results in **main( )**.

(c) Write a function that receives marks received by a student in 3 subjects and returns the average and percentage of these marks. Call this function from **main( )** and print the results in **main( )**.

[G] What would be the output of the following programs:

(a)
```
main( )
{
    int  i = 5, j = 2 ;
```

```
        junk ( i, j ) ;
        printf ( "\n%d %d", i, j ) ;
    }
    junk ( int  i, int  j )
    {
        i = i * i ;
        j = j * j ;
    }
```

(b)
```
    main( )
    {
        int  i = 5, j = 2 ;
        junk ( &i, &j ) ;
        printf ( "\n%d %d", i, j ) ;
    }
    junk ( int  *i, int  *j )
    {
        *i = *i * *i ;
        *j = *j * *j ;
    }
```

(c)
```
    main( )
    {
        int  i = 4, j = 2 ;
        junk ( &i, j ) ;
        printf ( "\n%d %d", i, j ) ;
    }
    junk ( int  *i, int  j )
    {
        *i = *i * *i ;
        j = j * j ;
    }
```

(d)
```
    main( )
    {
        float  a = 13.5 ;
        float  *b, *c ;
        b = &a ;  /* suppose address of a is 1006 */
```

```
    c = b ;
    printf ( "\n%u %u %u", &a, b, c ) ;
    printf ( "\n%f %f %f %f %f", a, *(&a), *&a, *b, *c ) ;
}
```

[H] Point out the errors, if any, in the following programs:

(a)
```
main( )
{
    int  i = 135, a = 135, k ;
    k = pass ( i, a ) ;
    printf ( "\n%d", k ) ;
}
pass ( int  j, int  b )
int  c ;
{
    c = j + b ;
    return ( c ) ;
}
```

(b)
```
main( )
{
    int  p = 23, f = 24 ;
    jiaayjo ( &p, &f ) ;
    printf ( "\n%d %d", p, f ) ;
}
jiaayjo ( int  q, int  g )
{
    q = q + q ;
    g = g + g ;
}
```

(c)
```
main( )
{
    int  k = 35, z ;
    z = check ( k ) ;
    printf ( "\n%d", z ) ;
}
```

```
check ( m )
{
    int  m ;
    if ( m > 40 )
        return ( 1 ) ;
    else
        return ( 0 ) ;
}
```

(d)
```
main( )
{
    int  i = 35, *z ;
    z = function ( &i ) ;
    printf ( "\n%d", z ) ;
}
function ( int  *m )
{
    return ( m + 2 ) ;
}
```

[I] What would be the output of the following programs:

(a)
```
main( )
{
    int  i = 0 ;
    i++ ;
    if ( i <= 5 )
    {
        printf ( "\nC adds wings to your thoughts" ) ;
        exit( ) ;
        main( ) ;
    }
}
```

(b)
```
main( )
{
    static int  i = 0 ;
    i++ ;
```

```
    if ( i <= 5 )
    {
        printf ( "\n%d", i ) ;
        main( ) ;
    }
    else
        exit( ) ;
}
```

[J] Attempt the following:

(a) A 5-digit positive integer is entered through the keyboard, write a function to calculate sum of digits of the 5-digit number:

(1) Without using recursion
(2) Using recursion

(b) A positive integer is entered through the keyboard, write a program to obtain the prime factors of the number. Modify the function suitably to obtain the prime factors recursively.

(c) Write a recursive function to obtain the first 25 numbers of a Fibonacci sequence. In a Fibonacci sequence the sum of two successive terms gives the third term. Following are the first few terms of the Fibonacci sequence:

1 1 2 3 5 8 13 21 34 55 89...

(d) A positive integer is entered through the keyboard, write a function to find the binary equivalent of this number using recursion.

(e) Write a recursive function to obtain the running sum of first 25 natural numbers.

(f) Write a C function to evaluate the series

$$\sin(x) = x - (x^3 / 3!) + (x^5 / 5!) - (x^7 / 7!) + \cdots$$

to five significant digits.

(g) Given three variables **x**, **y**, **z** write a function to circularly shift their values to right. In other words if x = 5, y = 8, z = 10 after circular shift y = 5, z = 8, x =10 after circular shift y = 5, z = 8 and x = 10. Call the function with variables **a**, **b**, **c** to circularly shift values.

(h) Write a function to find the binary equivalent of a given decimal integer and display it.

(i) If the lengths of the sides of a triangle are denoted by **a**, **b**, and **c**, then area of triangle is given by

$$area = \sqrt{S(S-a)(S-b)(S-c)}$$

where, S = (a + b + c) / 2

(j) Write a function to compute the distance between two points and use it to develop another function that will compute the area of the triangle whose vertices are **A(x1, y1)**, **B(x2, y2)**, and **C(x3, y3)**. Use these functions to develop a function which returns a value 1 if the point **(x, y)** lines inside the triangle ABC, otherwise a value 0.

(k) Write a function to compute the greatest common divisor given by Euclid's algorithm, exemplified for J = 1980, K = 1617 as follows:

	1980 – 1 * 1617 = 363
1980 / 1617 = 1	
1617 / 363 = 4	1617 – 4 * 363 = 165
363 / 165 = 2	363 – 2 * 165 = 33
5 / 33 = 5	165 – 5 * 33 = 0

Thus, the greatest common divisor is 33.