5. FUNCTIONS AND PROGRAM STRUCTURE

Functions form the cornerstone of C programming. As you expand your programming skills, your programs will take on a modular appearance when you begin programming with functions. You do all C programming within a function. This is because all programs must include ***main***, which is itself a function. If you have programmed in other languages, you will find C functions similar to modules in other languages. **Pascal** uses *procedures* and *functions*, **FORTRAN** uses just *functions*, and **assembly language** uses just *procedures*. How functions work determines to a large degree the efficiency, readability, and portability of C program code.

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

Function declaration and definition is the area where the ANSI standard has made the most visible changes to C. It is now possible to declare the types of arguments when a function is declared. The syntax of function definition also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically.

The standard clarifies the rules on the scope of names; in particular, it requires that there be only one definition of each external object. Initialization is more general: automatic arrays and structures may now be initialized.

The C preprocessor has also been enhanced. New preprocessor facilities include a more complete set of conditional compilation directives, a way to create quoted strings from macro arguments, and better control over the macro expansion process.

This chapter includes numerous C examples that illustrate how to write simple functions to perform specific tasks. The functions are short to make the concepts easier to understand and to prevent you from being lost in reams of code. Many of these examples use functions contained in the standard C libraries. If you learn to write good functions, you are well on your way to becoming a power programmer.

5.1 Function Style and Prototyping

C functions changed greatly during the ANSI standardization process. This new C standard is largely based on the function prototype used in C. As you read various articles, books, and magazines dealing with C, you will see many variations used to describe C functions, as programmers attempt (or don't attempt) to confirm to the ANSI C standard.

5.2 Function Prototyping

Function declarations begin with the C function prototype. The function prototype is simple and is included at the start of program code to notify the compiler of the type and number of arguments that a function will use. It also enforces a stronger type checking than was possible when C was not standardized.

Although other variations are legal, whenever possible you should use the function prototype form that is a replication of the function's declaration line. For example,

return_type function_name (argument_type(s) argument_name(s)) ;

The function can be of type **void**, **int**, **float**, and so on. The ***return_type*** gives this specification. The ***function_name*** is any meaningful name you choose to describe the function. If any information is passed to the function, you should give an ***argument_type*** followed by an ***argument_name***. Argument types may also be of type **void**, **int**, **float**, and so on. You can pass many values to a function by repeating the argument type and name separated by a comma. It is also correct to list just the argument type, but that prototype form is not used as frequently.

The function itself is an encapsulated piece of code that usually follows the **main** function definition. The function can take on the following form :

```
return_type  function_name ( argument_types and names )

{
        .
        .
        ( data declarations and body of function )
        .
        return( ) ;

}
```

5.3 The Return Statement

Parameters are one way functions can communicate; return values are another. We've used **return** to terminate a function and return a value to its caller.

```
return ( expression ) ;
```

But we can also use **return** to exit a function without returning a value.

```
return ;
```

return without an expression is similar to simply dropping off the function's end. Of course, we only use this form in functions that return void.

Notice that the first line of the function is identical (except for the missing ;) to the prototype that is listed at the beginning of a program. An actual function prototype and function, used in a program, is shown in the following C example :

```
/*   C program to illustrate function prototyping   */

        # include < stdio.h >

        int  additor ( int  a , int  b ) ;                  /* function prototype */

        void main (void)
        {
                int x , y , s ;

                printf (" Input two integers you want add \n ") ;
                scanf ("%d%d" , &x , &y ) ;

                s = additor( x , y ) ;

                printf (" Sum of the two numbers is = %d\n " , s ) ;

        }


        int  additor ( int  a , int  b )                    /* function declaration */
```

```
{
        int c ;

        c = a + b ;
        return (c) ;                        /* function return type */
}
```

The function is called **additor**. The prototype status that the function will accept two **int** arguments and return an **int** type. Actually, the ANSI standard suggests that every function be prototyped in a separate header file. As you might guess, this is how header files are associated with their appropriate C libraries. For simple programs, you can include the function prototype within the body of the program.

Eg : **/*** A C program demonstrating the functions ***/**

```
# include < stdio.h >

double Add ( float , float ) ;
double Sub ( float , float ) ;
double Mul ( float , float ) ;
double Div ( float , float ) ;
void Error (void ) ;

void main ( void )
{
        float Value1, Value2 ;
        char Operator ;

printf ("Enter your expression of the form value  operator  value \n" );
scanf ("%f%c%f" , &Value1, &Operator, &Value2 ) ;

        switch ( Operator )
        {
                case  '+' :  printf ("%lf \n" , Add(Value1,Value2) ) ;
                        break ;

                case '-' : printf ("%lf \n" , Sub(Value1,Value2) ) ;
                break ;

                case '*' :  printf ("%lf \n" , Mul(Value1,Value2) ) ;
                break ;

                case  '/' :  if ( Value2 != 0 )
                                printf ("%lf \n" , Div(Value1,Value2 ) );
```

```
                        else
                            Error ( ) ; break ;

                default   :  printf (" Unknown Operator \n " ) ; break ;
        }
}


double Add ( float Value1 , float Value2 )
{
        return ( Value1 + Value2 ) ;
}


double Sub ( float Value1 , float Value2 )
{
        return ( Value1 - Value2 ) ;
}


double Mul ( float Value1 , float Value2 )
{
        return ( Value1 * Value2 ) ;
}


double Div ( float Value1 , float Value2 )
{
        return ( Value1 / Value2 ) ;
}


void Error ( void )
{
        printf ( " Warning : Division by Zero \n " ) ;
}
```

5.4 Call-By-Value and Call-By-Reference

In the previous example, arguments were passed by value to the functions. When variables are passed in this manner, a copy of the variable's value is actually passed to the function. Since a copy is passed, the variable in the calling program is not altered. Calling a function by ***value*** is a popular means of passing information to a function and is the default method in C. The major

limitation to the ***call-by-value*** technique is that typically only one value is returned by the function.

In a ***call-by-reference***, the address of the argument, rather than its value, is passed to the function. This approach requires less program memory than a ***call-by-value***. When you use a ***call-by-reference***, the variables in the calling program can be altered. Additionally, more than one value can be returned by the function; but more on that later.

The next example uses the **additor** function from the previous section. The arguments are now passed as a ***call-by-reference***. In C, you achieve a ***call-by-reference*** by using a pointer as an argument.

```
/*   C program to illustrate call-by-reference   */

        # include < stdio.h >

        int  additor ( int  *a , int  *b ) ;              /* function prototype */

        void main (void)
        {
                int x , y , s ;

                printf (" Input two integers you want add \n ") ;
                scanf ("%d%d" , &x , &y ) ;

                s = additor( &x , &y ) ;

                printf (" Sum of the two numbers is = %d\n " , s ) ;

        }


        int  additor ( int  *a , int  *b )                /* function declaration */
        {
                int c ;

                c = *a + *b ;
                return (c) ;                              /* function return type */
        }
```

In C, you can use variables and pointers as arguments in function declarations. A **call-by-reference** is a favorite method of passing array information to a function.

5.5 Storage Classes and Functions

In C every variable declaration specify the **type**, **name**, **optional initialization** and **storage class** of an object. C supports four storage class specifiers :

- **auto**
- **register**
- **static**
- **extern**

The storage class precedes the variable's declaration and instructs the compiler how the variable should be stored.

auto variables

Declarations of variables within blocks are implicitly of storage class automatic. The keyword **auto** can be used to explicitly specify the storage class.

eg. auto int a , b , c ;

Storage class is automatic by default, the keyword **auto** is seldom used.

Despite their name, **auto** variables are not automatically initialized to zero. Instead, they simply start with whatever was previously in the memory location allocated for them-usually garbage. Conveniently, we can initialize a variable when we declare it simply by following its name with an equals sign and an arbitrary expression. This expression is evaluated whenever the function is called and the result assigned to the declared variable.

register variables

The storage class **register** tells the compiler that the associated variables should be stored in high speed memory registers. The storage class defaults to **auto** whenever the compiler cannot allocates an appropriate physical register. Typically, the compiler has only a few such registers available. Many of these are required for system use and cannot be allocated otherwise.

Basically, the use of storage class **register** is an attempt to improve execution speed. Loop variables and function parameters are commonly used as **register** variables.

eg. register int i ;

extern variables

One method of transmitting information across blocks and functions is to use external variables. When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is **extern**. The **extern** storage class specifier declares a reference to a variable defined elsewhere. You can use an **extern** declaration to make visible above its definition in the same source file. The variable is visible throughout the remainder of the source file in which the declared reference occurs. The keyword **extern** is used to tell the compiler to "***look for it elsewhere, either in the file or in some other file***".

eg. extern int a = 0, b = 2, c = 3 ;

You can use the keyword **extern** in function definitions as well. Functions declared as **extern** are visible throughout all source files that make up the program (unless you later redeclare such a function as **static**). Any function can call an **extern** function**. Function declarations that omit the storage class specifier are *extern* by default**.

```
eg.     extern  double  sin (double x)
        {

                ....

        }
```

static variables

You can define a variable at the external level only once within a source file. If you give the **static** storage class specifier, you can define another variable with the same name and the **static** storage class specifier in a different source file. Since each **static** definition is visible only within its own source file, no conflict occurs.

Static declarations have two important and distinct uses. The more elementary use is to allow a local variable to retain its previous value when the block is re-entered. This is in contrast to ordinary **auto** variables, which loose their value upon block exit and must be re-initialized. A variable declared at the internal level with the **static** storage class specifier has a global lifetime but is visible only within the block in which it is declared. You can initialize a **static** variable with a constant expression. **It is initialized to 0 by default.**

The second and more subtle use of **static** is in connection with external declarations. At first glance, **static external** variables seem unnecessary. External variables already retain their values across block and function exit. The difference is that **static extern** variables are scope restricted

external variables. The scope is the remainder of the source file in which they are declared. Thus they are unavailable to functions defined earlier in the file or to functions defined in other files, even if these functions attempt to use the **extern** storage class keyword.

Eg. Following C programs illustrate storage classes

(a)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;                    /* function prototype */

void main (void)
{
        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

}


int  additor ( int  a , int  b )                      /* function declaration */
{
        auto int c = 0 ;

        c = a + b + c ;

        return (c) ;                                  /* function return type */
}
```

(b)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;                    /* function prototype */

void main (void)
{
        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;
```

```
        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

}


int  additor ( int  a , int  b )                /* function declaration */
{
        extern int c = 0 ;

        c = a + b + c ;

        return (c) ;                            /* function return type */
}
```

(c)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;              /* function prototype */

int c = 0 ;

void main (void)
{
        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

}


int  additor ( int  a , int  b )                /* function declaration */
{
        extern int c = 0 ;

        c = a + b + c ;
```

```
        return (c) ;                        /* function return type */
}
```

(d)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;          /* function prototype */

int c = 0 ;

void main (void)
{
        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

}

int  additor ( int  a , int  b )            /* function declaration */
{
        auto int c = 0 ;

        c = a + b + c ;

        return (c) ;                        /* function return type */
}
```

(e)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;          /* function prototype */

void main (void)
{
        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;
```

```
        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

}


int  additor ( int  a , int  b )                /* function declaration */
{
        static int c = 0 ;

        c = a + b + c ;

        return (c) ;                            /* function return type */
}
```

(f)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;              /* function prototype */

void main (void)
{
        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

}


static int c = 0 ;

int  additor ( int  a , int  b )                /* function declaration */
{

        c = a + b + c ;
```

```
        return (c) ;                        /* function return type */
}
```

(g)

```
# include < stdio.h >

int  additor ( int  a , int  b ) ;          /* function prototype */

void main (void)
{

        int x , y , s ;

        printf (" Input two integers you want add \n ") ;
        scanf ("%d%d" , &x , &y ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        s = additor( x , y ) ;
        printf (" Sum of the two numbers is = %d\n " , s ) ;

        display( ) ;

}


static int c = 0 ;

int  additor ( int  a , int  b )            /* function declaration */
{

        c = a + b + c ;

        return (c) ;                        /* function return type */
}


void display(void)
{
        printf( "Value of c = %d\n" , c ) ;
}
```

(h) Move `static int c = 0` declaration statement in (g) to beginning of function additor.

5.6 Programming Problems involving Scope

You may experience unexpected program results when using variables with different scope levels. For example, you can use a variable of the same name with both **file** and **local** scopes. The scope rules state that the variable with **local** scope (called a local variable) will take precedence over the variable with **file** scope (called a global variable). However, there are some problem areas that you might encounter in programming.

An Undefined Symbol in a C Program

In the following example, four variables are given a local scope within the function **main**. Copies of the variables ***a*** and ***b*** are passed to the function **multiplier**. This does not violate scope rules. However, when the **multiplier** function attempts to use the variable ***c***, it cannot find the variable because the scope of the variable was to **main** only.

Eg :

```
# include <stdio.h>

int multiplier( int x, int y ) ;

void main( )
{
        int  a = 5,  b = 9,  c = 4,  d ;

        d = multiplier( a , b ) ;

        printf("The product is  =  %d \n", d ) ;
}

int multiplier( int x , int y )
{
        int  z ;

        z = x * y * c ;

        return( z ) ;
}
```

The compiler issues a warning and an error message. It first warns you that the ***c*** variable is never used within the function and then issues the error message that ***c*** has never been declared in the function **multiplier**. One way around this problem is to give ***c*** a file scope.

Use a Variable with File Scope

In this example, the variable ***c*** is given a file scope. If you make ***c*** global to the whole file, both **main** and **multiplier** can use it. Also note that both **main** and **multiplier** can change the value of the variable. If you want functions to be truly portable, you should not allow them to change program variables.

Eg :

```
# include <stdio.h>

int multiplier( int x, int y ) ;

int  c = 4 ;

void main( )
{
        int  a = 5,  b = 9,  d ;

        d = multiplier( a , b ) ;

        printf("The product is  =  %d \n", d ) ;
}

int multiplier( int x , int y )
{
        int  z ;

        z = x * y * c ;

        return( z ) ;
}
```

This program will compile correctly and print the product 180 to the screen.

Overriding a Variable with File Scope by a Variable with Local Scope

The scope rules state that variables with both file and local scope will use the local variable value over the global value.

Eg :

```
# include <stdio.h>

int multiplier( int x, int y ) ;

int  c = 4 ;
```

```
void main( )
{
        int  a = 5,  b = 9,  d ;

        d = multiplier( a , b ) ;

        printf("The product is  =  %d \n", d ) ;
}

int multiplier( int x , int y )
{
        int  z ;

        int  c = 2 ;

        z = x * y * c ;

        return( z ) ;
}
```

In this example, the variable ***c*** has both file and local scope. When ***c*** is used within the function **multiplier**, the local scope takes precedence and the product of 5 * 9 * 2 = 90 is returned.

Ex : What would be the output of the following programs ?

```
# include <stdio.h>

int  a = 1 , b = 2 , c = 3 ;
int  Add( void ) ;

void  main( )
{
        printf( "%3d\n" , Add( ) ) ;

        printf( "%3d%3d%3d\n" , a , b , c ) ;
}
```

(a)

```
int  Add( void )
{
        int  a , b , c ;
        a = b = c = 4 ;

        return( a+b+c ) ;
}
```

(b)
```
int  Add( void )
{
        int   b , c ;
        a = b = c = 4 ;

        return( a+b+c ) ;
}
```

(c)
```
int  Add( void )
{
        a = b = c = 4 ;

        return( a+b+c ) ;
}
```

Eg : For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the external level. The definition can be in any of the source files that form the program. The following C program demonstrates the use of the **extern** keyword:

/* Source FileA */

```
# include < stdio.h >

void function_a (void) ;
void function_b (void) ;

extern int int_value ;

void main(void)
{
        int_value++ ;

        printf (" (1). First Value = %d \n " , int_value ) ;

        function_a( ) ;

}

int_value = 10 ;

void function_a(void)
{
        int_value++ ;
```

```
        printf (" (2). Second Value = %d \n " , int_value ) ;

        function_b( ) ;
}
```

/* Source FileB */

```
# include < stdio.h >

extern int int_value ;

void function_b(void)
{
        int_value++ ;

        printf (" (3). Third Value = %d \n " , int_value ) ;
}
```

Eg. A variable declared with the **extern** storage class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The internal **extern** declaration is used to make the external level variable definition visible within the block. The next program segment demonstrates these concepts :

```
# include < stdio.h >

void function_a(void) ;

int int_value1 = 1 ;

void main(void)
{
        extern int int_value1 ;              /* reference the int_value1 defined
                                                above */

        static int int_value2 ;              /* default initialization of 0,
                                                int_value2 only  visible */

        register int register_value = 0 ;    /* stored in a register (if available),
                                                initialized to 0 */

        int int_value3 = 0 ;                 /* default auto storage class,
                                                int_value3 initialized to 0 */
```

```
        printf ("%d%d%d%d",int_value1, int_value2, regiater_value, int_value3);
        /*  values are printed 1 , 0 , 0 , 0  */

        function_a( ) ;

}


void function_a(void)
{
        /* stores the address of the global variable int_value1 */
        static int *pointer_to_int_value1 = &int_value1 ;

        /* create a new local variable int_value1 making the global int_value1
        unreachable */
        int int_value1 = 32 ;

        /* new local variable int_value2 only visible within function_a */
        static int int_value2 = 2 ;

        int_value2 += 2 ;

        printf ("%d%d%d" , int_value1, int_value2, *pointer_to_int_value1 ) ;
        /*  the values printed are 32 , 4 , and 1 */

}
```

5.7 Recursion

The C language supports a capability known as ***recursive*** function. Recursive functions can be effectively used to succinctly and efficiently solve problems. They are commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem. One example might be in the evaluation of expressions containing nested sets of parenthesized expressions. Other common applications involve the searching and sorting of data structures called ***trees*** and ***lists***.

Recursive functions are most commonly illustrated by an example which calculates the factorial of a number. As you will recall, the factorial of a positive integer *n*, written *n*!, is simply the product of the successive integers 1 through *n*. The factorial of 0 is a special case and is defined equal to 1. So 5! is calculated as follows :

5! = 5 x 4 x 3 x 2 x 1

$= 120$

and

$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$

$= 720$

Comparing the calculation of 6! to the calculation of 5! you will observe that the former is equal to 6 times the latter; that is, 6! = 6 x 5!. In general case, the factorial of any positive integer *n* greater than 0 is equal to *n* multiplied by the factorial of n-1 :

$$n! = \begin{cases} n \times (n-1)! & n > 0 \\ 1 & n = 0 \end{cases}$$

The expression of the value of *n*! in terms of the value of (*n*-1)! is called a *recursive* definition, since the definition of the value of a factorial is based on the value of another factorial. In fact, we can develop a function which calculates the factorial of an integer *n* according to this recursive definition. Such a function is illustrated in the following program.

Eg: Recursive function to calculate the factorial of a positive integer

```
# include < stdio.h >

long int Factorial( int ) ;

void main(void)
{
        int j ;
        clrscr( ) ;

        for ( j = 0 ; j < 11 ; j++ )
                printf(" %2d  =  %ld \n" , j , Factorial( j ) ) ;
}


long int Factorial( int n )
{
        long int Result ;

        if ( n == 0 )

                Result = 1 ;
```

```
        else
                Result = n * Factorial( n-1 ) ;
        return( Result ) ;
}
```

The fact that the ***Factorial*** function includes a call to itself makes this function recursive. Let's see what happens in the case when the function is called to calculate the factorial of 3, for example. When the function is entered, the value of the formal parameter *n* will be set to 3. Since this value is not zero, the following program statement

```
Result = n * Factorial( n - 1 ) ;
```

will be executed, which, given the value of *n*, will be evaluated as

```
Result = 3 * Factorial( 2 ) ;
```

This expression specifies that the ***Factorial*** function is to be called, this time to calculate the factorial of 2. Therefore, the multiplication of 3 by this value will be left pending while ***Factorial(2)*** is calculated.

With the value of *n* equal to 2, the ***Factorial*** function will execute the statement

```
Result = n * Factorial( n - 1 ) ;
```

Which will be evaluated as

```
Result = 2 * Factorial( 1 ) ;
```

Once again, the multiplication of 2 by the factorial of 1 will be left pending while the ***Factorial*** function is called to calculate the factorial of 1.

With the value of *n* equal to 1, the ***Factorial*** function will once again execute the statement

```
Result = n * Factorial( n - 1 ) ;
```

Which will be evaluated as

```
Result = 1 * Factorial( 0 ) ;
```

When the ***Factorial*** function is called to calculate the factorial of 0, the function will set the value of ***Result*** to 1 and ***return***, thus initiating the evaluation of all of the pending expressions.

In summary, the sequence of operations that is performed in the evaluation of factorial(3) can be conceptualized as follows :

```
Factorial( 3 )  =   3  *  Factorial( 2 )
                =   3  *  2  *  Factorial( 1 )
                =   3  *  2  *  1  *  Factorial( 0 )
                =   3  *  2  *  1  *  1
                =   6
```

Eg: Recursive function to calculate the factorial of a integer

```
# include < stdio.h >

long int Trace_Factorial( int ) ;

void main(void)
{
        clrscr( ) ;

        printf(" %ld \n" , Trace_Factorial( 4 ) ) ;
}

long int Trace_Factorial( int n )
{
        long int Result ;

        printf(" Computing factorial %d \n " , n ) ;
```

```
        Result = ( n <= 1 ) ?  1  :  n  *  Trace_Factorial( n - 1 ) ;

        printf(" Computed factorial %d  =  %ld \n", n , Result ) ;

        return( Result ) ;

}
```

The program output is shown below :

```
Computing factorial  4
Computing factorial  3
Computing factorial  2
Computing factorial  1
Computed factorial  1  =  1
Computed factorial  2  =  2
Computed factorial  3  =  6
Computed factorial  4  =  24
24
```

Eg : The Fibonacci numbers are a famous mathematical sequence that can be defined recursively. For nonnegative integers

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n >= 2 \end{cases}$$

```
# include < stdio.h >

# define MAX  100

void main( void )
{
        int i , n ;
        int Fibonacci[Max] ;
        Fibonacci[0] = 0 ;
        Fibonacci[1] = 1 ;

        printf(" Enter the number \n " ) ;
        scanf("%d" , &n ) ;

        for( i = 2 ; i < n ; i++ )
```

```
                Fibonacci[i] = Fibonacci[i-1] + Fibonacci[i-2] ;

        for( i = 0 ; i < n ; i++ )

                printf(" Fibonacci[%d] = %d\n " , i , Fibonacci[i] ) ;

}
```

Eg : Function to compute the square root of a number. If a negative argument is passed, then a message is displayed and -1.0 is returned.

```
float Square_Root ( float x )
{
        float epsilon = 0.00001 ;
        float guess   = 1.0 ;

        if ( x < 0 )
        {
                printf(" Negative argument to square_root \n " ) ;
                return( -1.0 ) ;
        }

        while ( absolute_value ( guess * guess - x ) >= epsilon )

                guess = ( x / guess + guess ) / 2.0 ;

        return( guess );

}


float absolute_value ( float x )
{
        if ( x < 0 )

                x = -x ;

        return ( x ) ;
}
```

Eg : A program that computes x^y for integers x and y

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

```
double power ( int x , int y ) ;

void main ( void )
{
        int x , y ;

        while( printf("Enter x and y :") , scanf ("%d%d", &x ,&y) == 2 )

                printf("%d to the %d is = %Lf \n" , x , y , power( x , y ) ) ;
}


double power ( int x , int y )
{
        double p = 1.0 ;

        if ( y >= 0 )

                while ( y-- )                   /* positive powers */

                        p *= x ;
        else

                while ( y++ )                   /* negative powers */

                        p /= x ;

        return ( p ) ;

}
```

5.8 Character Input and Output

Just as there are prewritten routines for formatted input and output of numerical and string data, there are predefined functions for character input and output. The character-equivalent functions of *scanf* and *printf* are **getchar** and **putchar**, respectively.

getchar and putchar

getchar takes no arguments and returns as its value the integer representation of a single character-the next character in the input stream. Like *scanf*, **getchar** returns the special value EOF if the end-of-file character is entered. As we saw earlier, EOF is usually -1, a value that does not represent a legal character.

Similarly, **putchar** takes an integer that represents a legal character, and writes it as a character to the standard output. Anomalous results occur if the number does not represent a legal character, although most machines simply use the least significant bits (equivalent to the local character representation). It's your responsibility to provide a value that represents a legitimate character. **putchar** does not return a value and should not fail, although it can fail if the character is being written to a file.

As a result of the need for an "***extended***" character set (that is, one that includes all of the machine's characters, plus EOF), we usually deal with variables declared as **int**s when reading character data. We can see this from program **display.c** given below :

```
# include < stdio.h >

void main( void )
{
        int C;

        while ( ( C =getchar( ) ) != EOF )

                putchar( C ) ;
}
```

that echoes its input to its output, one character at a time.

The program itself simply reads characters using **getchar** and writes them with **putchar**, stopping when **getchar** returns EOF. We have included *stdio.h* because **getchar** and **putchar**, as well as EOF, are defined there. Because of the need for the extended character set, the variable *C*, which holds the character being read or written, is declared as an **int**. Note that reading the

character and assigning it to *C* occur within the loop's test. taking advantage of the assignment operator's ability to return the value that it has assigned.

It turns out that **getchar** is buffered (as is *scanf*). In other words, when we read from the keyboard, the operating system collects characters in a special location until we type a carriage return or hit the enter key. And it does something similar when we read from a file, except that it reads a larger chunk of characters each time. All **getchar** does is return the next character in the buffer. This buffering is also done when we do output to a file. The operating system collects characters in a different location and writes them to the file in large groups. Why buffer at all ? Because it allows us to use backspace to edit out input, and because it makes input and output more efficient.

We can take advantage of this buffering to extend our character-copying program to perform more complex input transformations. The program given below prints each line in its input preceded with a line number. Like above program, this program reads one character at a time until it reaches the end of the input, writing each character onto its output. But this program also remembers the last character read. When a character is to be printed, if the preceding character was a newline(\n), it is the beginning of a new line, and a line number is printed.

Eg : A program to line number its input

```
# include < stdio.h >

void main( void )
{
        int  C ,                        /* current and  */
             lastch,                    /* previous charactes */
             lineno ;                   /* lines printed so far */

        lineno = 0 ;

        for ( lastch = '\n' ;  ( C = getchar( ) ) != EOF ;  lastch = C )
        {
                if ( lastch == '\n' )
                {
                        lineno++;

                        printf("%6d", lineno ) ;

                }
                putchar( C ) ;

        }
        return ;
```

```
}
```

Eg : Write a C program to count the number of lines input

```
# include < stdio.h >

void main( void )
{
        int C , Linecount = 0 ;

        while( ( C = getchar( ) ) != EOF )

                if ( C == '\n' )

                        Linecount++;

        printf(" Number of Lines = %d\n " , Linecount ) ;
}
```

Ex : Write a program that eliminates all blank lines from its input.

5.9 Unbuffered Character Input

Because **getchar** buffers its input, it is not well suited for interactive programs such as editors or menu-driven interfaces. These programs want to read a character as soon as the user types it-they don't want to wait for a carriage return. An interactive screen editor, for example, shouldn't make the user type a carriage return after entering an editor command. And an interactive data base interface shouldn't force the user to hit ENTER after making a menu selection. In fact, when we do console I/O (reading input from the keyboard or writing output to the display) we generally don't want buffering.

By default, console output is unbuffered. And luckily Turbo C provides two special functions, **getche** and **getch**, that let us do unbuffered input. To use these functions we include *conio.h*.

getche is identical to **getchar** except that it doesn't wait for a carriage return. That is, we get the character when the user types it. The "*e*" at the end of its name stands for *echo*; **getche** choes the character as it's typed (as does **getchar**). **getch** is like **getche** except that it does not echo the character.

We use getche in following program, which contains a function *yesorno* that obtains a "*yes*" or "*no*" response. *yesorno* uses *getche* to read the next input character, and then determines whether it is a 'Y' or an 'N'. An invalid character makes *yesorno* print an error message and repeat the process; otherwise, it returns a 1 for a "yes" and a 0 for a "no".

Eg : A first version of a program and function to get a Yes or No answer from the user

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

int yesorno( void ) ;

void main( void )
{
        int answer ;

        printf(" Do you like Turbo C ?  " ) ;

        if ( (answer = yesorno( ) ) != 0 )

                printf(" We received a YES! \n " ) ;

        else

                printf(" We received a NO! \n " ) ;

        return ;
}


int yesorno( void )
{
        int C ;
        while ( ( C = getche( ) ) != 'Y'  &&  C != 'N' )
        {
                putchar('\n') ;

                putchar( " Please enter a Y or N :  " ) ;
        }

        putchar('\n') ;

        return( C == 'Y') ;
}
```

We use **getch** in a second version of *yesorno*, showm below. This one doesn't echo the character the user types, but instead rings a bell if the user types an inappropriate character.

Eg : A second version of a program and function to get a Yes or No answer from the user, and it displays the entire YES or NO, not just its first letter. The function beeps if given an incorrect answer.

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

int yesorno( void ) ;

void main( void )
{
        int answer ;

        printf(" Do you like Turbo C ?  " ) ;

        if ( (answer = yesorno( ) ) != 0 )

                printf(" YES \n " ) ;

        else

                printf(" NO \n " ) ;

        return ;
}



int yesorno( void )
{
        int C ;
        while ( ( C = getche( ) ) != 'Y'  &&  C != 'N' )
                putchar('\a') ;

        return( C == 'Y') ;
}
```

How do we decide which input function to use? We use **getchar** when we don't care whether we obtain the character as soon as the user types it, or when our program's input is likely to be redirected. We use **getche** when we need the character as soon as the user types it and we want it echoed to the screen. And we use **getch** when we want the character right away but don't want it echoed.

Warning : Don't interweave calls to **getchar** with calls to **getche** and **getch**. Why not? Because all of these functions read from the standard input, and only **getchar** uses a buffer, following a **getchar** witha **getch** can cause any characters in that buffer to be ignored.

5.10 Character Testing Functions

When we read a character we often need to know what type of character we have. Is it an uppercase letter-or lowercase? A digit? Is it printable-or is it a control character? And so on.

One way to find out this information is to check whether the character falls within a particular range of characters. For example, we can determine whether *C* is a lowercase letter with :

```
if ( C >= 'a'  &&  C<='z' )

        printf(" It is a lowercase letter \n ") ;
```

This tests whether *C*'s integer representation is between the integer representations of 'a' and 'z'. That works fine if the lowercase letters are contiguous within the local character set. Unfortunately, while this is true of ASCII, it is not true of EBCDIC. If we want our program to work regardless of the underlying character set, we need another method. Fortunately, C provides a set of functions (macros, actually) that we can use to perform these comparisons. These functions are listed below, and to use them we include ***ctype.h***.

Function	**Character Type**
isalpha	a letter (a - z , A - Z)
islower	a lowercase letter (a - z)
isupper	an uppercase letter (A - Z)
isalnum	a letter or digit (a - z , A - Z , 0 - 9)
isdigit	a digit (0 - 9)
isxdigit	a hexadecimal digit (0 - 9 , a - f , A - F)
isspace	a space, \t , \v , \f , \r , or \n
iscntrl	a control (0x00 - 0x1F) or delete (0x7F) character
ispunct	punctuation (isprint && !isalnum)

isgraph	displayable (not including space) (0x21 - 0x7E)
isascii	an ASCII character (0x00 - 0x7E)
isprint	printable (0x20 - 0x7E)

Each of these functions takes a character and returns a nonzero value if it falls into the given class and zero if it doesn't. For example, we can use ***islower*** to rewrite the test above.

```
if ( islower( C ) != 0 )
        printf( " It is a lowercase letter \n " ) ;
```

And we can verify that a character is not an uppercase letter in a similar way.

```
if ( isupper( C ) == 0 )
        printf( " It is not a uppercase letter \n " ) ;
```

Actually, we can write these tests even more concisely. Earlier we mentioned that C considers an expression that evaluates to zero to be false, and any other expression to be true. So

```
if ( islower( C ) )
        printf( " It is a lowercase letter \n " ) ;
```

and

```
if ( isupper( C ) )
        printf( " It is not an uppercase letter \n " ) ;
```

are more compact versions of our earlier tests.

In addition to the functions for testing characters, there is also a small set of functions for converting characters. These functions are listed below, and each takes a character and returns a character.

Function	**Converts**
tolower	uppercase to lowercase (others unchanged)
toupper	lowercase to uppercase (others unchanged)

_tolower	uppercase to lowercase (must be uppercase)
_toupper	lowercase to uppercase (must be lowercase)
toascii	character to ASCII (clear high-order bits)

Eg : Program to print counts of various types of characters (spaces, digits, letters, punctuations, and so on)

```
# include < stdio.h >

void main( void )
{
        int C , spaces = 0 , letters = 0 , digits = 0 , puncts = 0 , others = 0 ;

        while ( ( C = getchar( ) ) != EOF )
        {
                if ( isspace( C ) )

                        spaces++ ;

                else if ( isalpha( C ) )

                        letters++ ;

                else if ( isdigit( C ) )

                        digits++ ;

                if ( ispunct( C ) )

                        puncts++ ;

                else

                        others++ ;

        printf("%3d%3d%3d%3d%3d",spaces,letters,digits,puncts,others);

}
```

5.11 Access Modifiers

The **const** and **volatile** modifiers are new to C and C++. They were added by the ANSI C standard to help identify variables that will never change (**const**) and variables that can change unexpectedly (**volatile**).

const Modifier

At times, you may need to use a value that does not change throughout the program. Such a quantity is called a *constant*. For example, if a program deals with the area and circumference of a circle, it will frequently use the constant value *pi = 3.14159*. In a financial program, an interest rate might be a constant. In such cases, you can improve the readability of the program by giving the constant a descriptive name.

Using descriptive names can also help prevent errors. Suppose that you use a constant value (not a constant variable) at many points throughout the program. Suppose also that you type the wrong value at one or more of these points. If the constant has a name, a typographical error would then be detected by the compiler because you probably didn't declare the incorrect value.

Suppose that you are writing a program that repeatedly uses the value π. You might think that you should declare a variable called *pi* with an initial value of 3.14159. However, the program should not be able to change the value of a constant. For instance, if you inadvertently wrote *pi* to the left of an equal sign, the value of *pi* would be changed, causing all subsequent calculations to be in error. C provide mechanisms that prevent such errors from occurring - that is, you can establish constants whose values cannot be changed.

In C, you declare a constant by writing **const** before the keyword (for instance, **int**, **float**, **double**) in the declaration. For example:

```
const int MAX = 9, INTERVAL = 15 ;
const float RATE = 0.7 ;
int index = 0, count = 10, object ;
double distance = 0.0, velocity ;
```

Because a constant cannot be changed, it must be initialized in its declaration. The **int** constants *MAX* and *INTERVAL* are declared with values 9 and 15, respectively. The constant *RATE* is of type **float** and has been initialized to 0.7. In addition, the **int** (nonconstant) variables *index*, *count*, and *object* have been declared. Initial values of 0 and 10 have been established for *index* and *count*, respectively. Finally, *distance* and *velocity* have been declared to be (nonconstant) variables of type **double**. An initial value of 0.0 has been established for *distance*.

You use constants and variables in the same way in a program. The only difference is that you cannot change the initial values assigned to the constants-that is, the constants are not *lvalues*; they cannot appear to the left of an equal sign.

Normally, the assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression that refers to a modifiable memory location.

Expressions that refer to memory locations are called ***lvalue expressions***. Expressions referring to modifiable locations are modifiable ***lvalues***. One example of a modifiable ***lvalue*** expression is a variable name declared without **const**.

define Constants

C provide another method for establishing constants-the **#define** compiler directive. Suppose that you have the following statement at the beginning of a program:

```
# define VOLUME 10
```

The form of this statement is **#define** followed by two strings of characters separated by spaces. When the program is compiled, several passes are made through it. First, the compiler preprocessor carries out the **#include** and **#define** directives. When the preprocessor encounters the **#define** directive, it replaces every occurrence of *VOLUME* in the source files with the number 10.

In general, when the preprocessor encounters a **#define** directive, it replaces every occurrence of the first string of characters (*VOLUME*) in the program with the second string of characters (10). Additionally, no value can be assigned to *VOLUME* because it has never been declared as a variable. As a result of the syntax, *VOLUME* has all the attributes of a constant. Note that the **#define** statement is not terminated by a semicolon. If a semicolon followed the value 10, every occurrence of *VOLUME* would be replaced with 10;. The directive replaces the first string with *everything* in the second string.

The short programs that have been discussed so far would usually be stored in a single file. If a statement such as the **#define** for *VOLUME* appeared at the beginning of the file, the substitution of 10 for *VOLUME* would take place throughout the program. Later, you will learn how to divide a program into many subprograms, with each subprogram being divided into separate files. Under these circumstances, the **#define** compiler directive would be effective only for the single file in which it is written.

You have learned two methods for defining constants: the keyword **const** and the **#define** compiler directive. In many programs, the action of each of these two methods is essentially the same. On the other hand, the use of the modifier keyword **const** results in a “variable” whose value cannot be changed. Later you will see how you can declare variables in such a way that they exist only over certain regions of a program. The same can be said for constants declared with the keyword **const**. Thus, the **const** declaration is somewhat more versatile than the **#define**

directive. Also, the **#define** directive is found in standard C and is therefore already familiar to C programmers.

volatile Modifier

The **volatile** keyword signifies that a variable can unexpectedly change because of events outside the control of the program. For example, the following definition indicates that the variable ***timer*** can have its value changed without the knowledge of the program:

```
volatile  int  timer ;
```

You need a definition like this, for example, if ***timer*** is updated by hardware that maintains the current clock time. The program that contains the variable ***timer*** could be interrupted by the time-keeping hardware and the variable ***timer*** changed.

You should declare a data object **volatile** if it is a memory-mapped device register or a data object shared by separate processes, as would be the case with a multitasking operating environment.

const and volatile

You can use the two modifiers **const** and **volatile** with any other data types-for example, **char** and **float-**as well as with each other. The following definition

```
const  volatile  constant_times ;
```

specifies that the program does not intend to change the value in the variable *constant_timer*. However, the compiler is also instructed, because of the **volatile** modifier, to make no assumptions about the variable's value from one moment to the next. Therefore, the compiler first issues an error message for any line of source code that attempts to change the value of the variable *constant_timer* from inside loops, since an external process can also be updating the variable while the program is executing.

pascal, cdecl, near, far, and huge Modifiers

The modifiers **pascal** and **cdecl** are used most frequently in advanced applications. C allows you to write programs that can easily call other routines written in different languages. The opposite

of this also holds true. For example, you can write a Pascal program that calls a C routine. **When you mix languages in this way, you have to consider two very important issues: identifier names, and the way that parameters are passed**.

When C compiles your program, it places all of the program's global identifiers(functions and variables) into the resulting object code file for linking purposes. By default, the compiler saves those identifiers using the case in which they were defined (upper, lower, or mixed). Additionally, the compiler appends an underscore to the front of the identifier (you can turn this feature off with the -u option). Since Turbo C integrated linking is case sensitive by default, any external identifiers that you declare in your program are also assumed to be in the same form, with a preceding underscore and the same spelling and case as defined.

pascal

The Pascal language uses a different calling sequence than C. Pascal (along with FORTRAN) passes function arguments from left to right and does not allow variable-length argument lists. In Pascal, the called function removes the arguments from the stack. (In C, the invoking function does so when control returns from the invoked function.)

A C program can generate this calling sequence in one of two ways: It can use the compile-time switch -p which makes the Pascal calling sequence the default for all enclosed calls and function definitions; it can also override the default C calling sequence explicitly by using the **pascal** keyword in the function definition.

As mentioned, when C generates a function call, by default it precedes the function name with an underscore and declares the function as external. It also preserves the case of the name. However, with the **pascal** keyword, the underscore is not used and the identifier (function or variable) is converted to all uppercase.

The following code segment demonstrates how to use the **pascal** keyword on a function :

```
int pascal myfunction( int value1, long value2, double value3 )
{
        .
        .
        .
        .
}
```

Of course, you can also give variables a Pascal convention, as shown here :

```
#define MAX 100
```

```
int pascal myfunction( int value1, long value2, double value3 )
{
        .
        .
        .
        .
}

int pascal mytable[MAX] ;

void main(void)
{
        int a = 5 , result ;
        long b = 2.345 ;
        double c = 7832.89901 ;

        result = myfunction( a , b , c ) ;

        return(0) ;

}
```

In this example, **mytable** has been globally defined with the **pascal** modifier. Function **main** also shows how to make an external reference to a Pascal function type. Since both functions **main** and **myfunction** are in the same source file, the function **myfunction** is global to **main**.

cdecl

If the -p compile-time switch was used to compile your C program, all function and variable references were generated matching the Pascal calling convention. However, you will sometimes want to guarantee that certain identifiers in your program remain case sensitive and retain the initial underscore. This is most often the case for identifiers being used in another C file.

To maintain this C compatibility (preserving the case and the leading underscore), you can use the **cdecl** keyword. When you use the **cdecl** keyword in front of a function, it also affects how the parameters are passed.

Note : All C functions defined in the header files, for example *stdio.h*, are of type **cdecl**. This ensures that you can link with the library routines, even when you are compiling by using the -p option. The following example was compiled by using the -p option and shows how you would rewrite the previous example to maintain C compatibility:

```
#define MAX 100

int cdecl myfunction( int value1, long value2, double value3 )
{
        .
        .
        .
        .
}

int cdecl mytable[MAX] ;

void main(void)
{
        int a = 5 , result ;
        long b = 2.345 ;
        double c = 7832.89901 ;
        extern int cdecl myfunction( ) ;

        result = myfunction( a , b , c ) ;

        return(0) ;

}
```

near, far, and huge

The three modifiers **near**, **far**, and **huge** affect the action of the indirection operator ∗. In other words, they modify pointer sizes to data objects. A **near** pointer is only 2 bytes long, a **far** pointer is 4 bytes long, and a **huge** pointer is also 4 bytes long.