6. ARRAYS

The C language provides a capability that enables the user to define a set of ordered data items known as an array. This chapter describes how arrays can be defined and manipulated in C. Arrays are indexed variables that contain many data items of the same type. Each array has one name, and you refer to the individual elements of the array by associating a subscript, or index, with the array name. In the C language, an array is not a basic type of data; instead, it is an aggregate type made up of other data types. In C, you can have an array of anything: characters, integers, floats, doubles, arrays, pointers, structures, and so on.

The Basic Properties of an Array

An array has four basic properties:

- The individual data items in the array are called ***elements***.
- All elements must be of the same data type.
- All elements are stored contiguously in the computer's memory, and the subscript (or index) of the first element is zero.
- The name of the array is a constant value that represents the address of the first element in the array

Because all elements are assumed to be of the same size, you cannot define arrays using mixed data types. If you did, it would be very difficult to determine where any given element was stored. The fact that all elements in an array are of the same size helps determine how to locate a given element. The elements are stored contiguously in the computer's memory (with the lowest address corresponding to the first element, and the highest address to the last element). In other words, there is no filler space between elements, and they are physically adjacent in the computer.

It is possible to have arrays within arrays-that is, multidimensional arrays. Actually, if an array element is a structure, other data types can exist in the array by existing inside the structure member.

Finally, the name of an array represents a constant value that cannot change during the execution of the program. For this reason, some forms of expressions that might appear to be valid are not allowed. You will eventually learn these subtleties.

6.1 Defining an Array

To define an array, write the array type, followed by a valid array name and a pair of square brackets enclosing a constant expression. The constant expression defines the size of the array. You cannot use a variable name inside the square brackets-in other words, you can't avoid specifying the array size until the program actually runs. The expression must reduce to a constant value, because the compiler has to know exactly how much storage space to reserve for the array. It is best to use defined constants to specify the size of the array, as in:

```
# define  MAX  10

int Name[MAX] ;
```

By using defined constants, you can ensure that subsequent references to the array will not exceed the defined array size. For instance, it is very common to use a defined constant as a terminating condition in a ***for*** loop that accesses array elements, as in this example:

```
# include <stdio.h>

# define MAX 20

void print_table( int [ ] , int , int ) ;

void main( void )
{
        int table[MAX]  ,  i  ;

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

                table[i] = i ;

        print_table( table , 0 , MAX - 1 ) ;     /* print the reversed array */

        return ;

}

void print_table( int a[ ] , int first , int last )
{

        for(  ;  first < last ; first++ )

                printf("%d\n" , a[first] ) ;
}
```

6.2 Initializing an Array

There are three methods for initializing arrays :

- By default when they are created. This only applies to global and static automatic arrays.
- Explicitly when they are created by supplying constant initializing data.
- During program execution by assigning or copying data into the array.

Initializing by Default

The ANSI C standard specifies that arrays are either *global* (defined outside of **main** and any other function) or *static automatic* (static, but defined after any opening brace) and will always be initialized to zero if no other initialization data is supplied. You can run the following program to make certain that your compiler meets this standard:

```
# include <stdio.h>
# define  MAX  10

int  Name[MAX] ;                    /*  a global array  */

void main( void )
{

        static int  B[MAX] ;        /*  a static array  */
                .
                .
                .

}
```

When you run the program, zeroes should verify that both array types are indeed automatically initialized.

Explicitly Initializing an Array

Just as you can define and initialize variables of type *int*, *char*, *float*, *double*, and so on, you can also initialize arrays. The ANSI C standard lets you supply initialization values for any array, global or otherwise, defined anywhere in a program. The following code segment illustrates how to define and initialize arrays :

```
int  Numbers[ ] = { 1 , 2 , 3 , 4 , 5 };

char  Letters[ ]  =  { ‘a’ ,  ‘b’ ,  ‘c’ ,  ‘d’ , ‘e’ };

static int  New_Numbers[ ] = { 1 , 2 , 3 , 4 , 5 } ;

static float  Cost[ ] = { 12.5 , 17.8 , 22.4 , 24.5 , 19.75 } ;

static float  Sample[5] = { 12.5 , 17.8 , 22.4 } ;
```

will initialize the first 3 values of Sample to 12.5 17.8 and 22.4 and will set the remaining 2 elements to zero.

Initializing during program execution

```
# include <stdio.h>
# define MAX 10

void main( void )
{

        static int Values[MAX] ;
        int i ;

        for ( i = 0 ; i < MAX ; i++ )
                Values[i] = i * i ;

        for ( i = 0 ; i < MAX ; i++ )
                printf(“ Values[%d] = %d \n ” , i , Values[i] ) ;

}
```

The output of this program :

Values[0]	**0**
Values[1]	**1**
Values[2]	**4**
Values[3]	**9**
Values[4]	**16**
Values[5]	**25**
Values[6]	**36**
Values[7]	**49**
Values[8]	**64**
Values[9]	**81**

Unsized Array Initializations

Most compilers require either the size of the array or the list of actual values, but not both. For example, a program will frequently want to define its own set of error messages. You can do this in one of two ways. Here is one method :

```
char Error[22]  =  “University of Colombo” ;
```

This approach can become tedious-straining your eyes as you count the number of characters-and is very error prone. Here is an example of the second method :

```
char Error[ ]  =  “University of Colombo” ;
```

This method allows C to dimension the arrays automatically with unsized arrays. Whenever C encounters an array initialization statement and the array size is not specified, the compiler automatically creates an array large enough to hold all of the specified data.

An array with an empty size declaration and no list of values has a NULL length. If any declaration follows, the name of the NULL array refers to the same address, and storing values in the NULL array puts them in addresses allocated to *other* variables.

Also, unsized array initializations are not restricted to one-dimensional arrays. For multidimensional arrays, you must specify all but the leftmost dimension for C to index the array properly. With this approach, you can build tables of varying length and the compiler will automatically allocate enough storage.

Eg :

```
char  Name[ ] = “ABC” ;

printf(“%d” , Name[0] ) ;      ⇒     65

printf(“%c” , Name[0] ) ;      ⇒     A

printf(“%s” , Name ) ;⇒      ABC

printf(“%d” , Name[1] ) ;      ⇒     66

printf(“%c” , Name[1] ) ;      ⇒     B

printf(“%s” , Name + 1 ) ;     ⇒     BC

printf(“%d” , Name[2] ) ;      ⇒     67

printf(“%c” , Name[2] ) ;      ⇒     C
printf("%s" , Name + 2 ) ;      ⇒       C
```

Note :	Name = &Name[0]	and	Name[0] = ∗Name
	Name + 1 = &Name[1]	and	Name[1] = ∗(Name + 1)
	Name + 2 = &Name[2]	and	Name[2] = ∗(Name + 2)

6.3 Array Boundary Checking

C array types offer faster executing code at the expense of zero boundary checking. Remember, since C was designed to replace assembly language code, error checking was left out of the compiler to keep the code concise. Without any compiler error checking, you must be very careful when dealing with array boundaries. For example, the following program incites no complaints from the compiler, and yet it can change the contents of other variables or even crash the program :

```
# include < stdio.h >

# define LIMIT  10
# define TROUBLE  50

void main( void )
{
        int collide[LIMIT] , index ;

        for( index = 0 ; index < TROUBLE ; index++ )
                collide[index] = index ;

}
```

6.4 Arrays and Functions

Just like other C variables, arrays can be passed from one function to another. Because you must understand pointers to understand arrays as function arguments, the topic is covered in more detail in chapter "***Pointers***".

Array Arguments in C

Consider a function **total** that computes the sum of the array elements *array_value[0], array_value[1],......, array_value[n]*. Two parameters are required : an array parameter *array_value_received* to catch the array passed, and a parameter *current_index* to catch the index of the last item in the array to be totaled. Assuming that the array is an array of **int**s and that the index is also of type **int**, the parameters in **total** can be described as

```
int total( int array_value_received[ ] , int current_index )
```

The parameter declaration for the array includes square brackets to tell the function **total** that *array_value_received* is an array name and not the name of an ordinary parameter. Note that the number of cells is *not* enclosed within the square brackets. Of course, the simple parameter *current_index* is declared as previously described. Invoking the function is as easy as

```
result  =  total( array_value , actual_index ) ;
```

Passing the array *array_value* just involves entering its name as the argument. When passing an array's name to a function, you are actually passing the address of the array's first element. The expression

```
array_value
```

is already shorthand for

```
&array_value[0]
```

Technically, you can invoke the function **total** with either of the two following valid statements :

```
result = total( array_value , actual_index );

result = total( &array_value[0] , actual_index );
```

In either case, within the function **total** you can access every cell in the array.

When a function is going to process an array, the calling function includes the name of the array in the function's argument list. This means that the function receives and carries out its processing on the actual elements of the array, not on a local copy as in single-value variables where functions pass only their values.

When a function is to receive an array name as an argument, there are two ways to declare the argument locally : as an array or as a pointer. Which method you use depends on how the function processes the set of values. If the function steps through the elements with an index, the declaration must be an array with square brackets following the name. The size can be empty since the declaration does not reserve space for the entire array, just for the address where it begins. Having seen the array declaration at the beginning of the function, the compiler permits brackets with an index to appear after the array name anywhere in the function.

The following program declares an array of 10 elements and, after printing its values, calls in a function to determine the largest value in the array. To do this, it passes the array name and size of the function **find_largest**, which declares them as an array called *values*[]. The function then passes through the array, comparing each element against the largest value it has seen so far. Every time it encounters a bigger value, it stores that new value in the variable *largest_value*. At the end, it returns the largest value that it has detected for **main** to print.

Eg : A C program that demonstrates the passing of arrays.

```
# include < stdio.h >
# define SIZE 10

int find_largest( int [ ]  ) ;

void main( void )
{
        int scores[10] , maximum_score , i ;

        printf( “ Enter 10 scores : \n” ) ;

        for( i = 0 ; i < SIZE ; ++i )
                scanf( “%d” , &scores[i] ) ;

        maximum_score = find_largest(scores) ;

        printf(“Maximum score = %d \n”, maximum_value ) ;
}

int find_largest( int values[ ] )
{
        int largest_value , i ;

        largest_value = values[0] ;

        for( i = 1 ; i < SIZE ; ++i )

                if( values[i] > largest_value )

                        largest_value = values[i] ;
```

```
          return ( largest_value ) ;

     }
```

Eg : A C program to find the maximum of an array.

```
     # include < stdio.h >

     int find_largest( int [ ]  ,  int  ) ;

     void main( void )
     {
          static int Array1[5] = { 157 , -28 , -37 , 26 , 10 } ;
          static int Array2[7] = { 12 , 45 , 1 , 10 , 5 , 3 , 22 } ;

          printf("Array1 Maximum score = %d \n", find_largest(Array1,5) );

          printf("Array2 Maximum score = %d \n", find_largest(Array2,7) );

     }


     int find_largest( int values[ ] , int number_of_elements )
     {
          int largest_value , i ;

          largest_value = values[0] ;

          for( i = 1 ; i < number_of_elements ; ++i )

               if( values[i] > largest_value )

                    largest_value = values[i] ;

          return ( largest_value ) ;

     }
```

Eg : A modified C program to sort an array by ascending order.

```
     # include < stdio.h >
     # define SIZE 10
```

```
int sort_array( int [ ]  ) ;

void main( void )
{
        int scores[10] ,  i ;

        printf( " Enter 10 scores : \n" ) ;

        for( i = 0 ; i < SIZE ; ++i )
                scanf( "%d" , &scores[i] ) ;

        sort_array(scores) ;
}


int sort_array( int values[ ] )
{
        int  i , j , temp ;

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

                for( j = i+1 ; j < SIZE ; j++ )

                        if( values[j] > values[i] )
                        {

                                temp = values[i] ;
                                values[i] = values[j] ;
                                values[j] = temp ;
                        }

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

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

}
```

Eg : A C program to reverse an array in place and then print it.

```
# include <stdio.h>

# define MAX 20

void reverse_table( int [ ] , int  , int  ) ;
void print_table( int [ ] , int , int ) ;
void main( void )
{
        int table[MAX]  ,  i  ;

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

                table[i] = i ;

        reverse_table( table , 0 , MAX - 1 ) ;    /*  reverse the array  */
        print_table( table , 0 , MAX - 1 ) ;      /* print the reversed array */

}

void reserve_table( int a[ ] , int first , int last )
{
        int  temp ;

        for(  ;  first < last ; first++ , last-- )
        {
                temp = a[first] ;

                a[first] = a[last] ;

                a[last] = temp ;
        }
}

void print_table( int a[ ] , int first , int last )
{

        for(  ;  first < last ; first++ )

                printf("%d\n" , a[first] ) ;
}
```

6.5 Converting Characters into Numbers

We often want to read characters and convert them into integers. Our first thought is to use ***scanf***-but unfortunately it handles input errors poorly. An alternative is to read characters one at a time, and to convert them into a single integer ourselves.

Eg : A C program which converts a string of digits into its numeric equivalent - version 1

```
# include < stdio.h >

void main( void )
{
        char  S[ ]  =  “12467” ;

        printf(“%ld \n”, atoi( S ) ) ;
}


long int atoi( char S[ ] )
{
        int  i ;

        long int n = 0 ;

        for( i = 0 ; S[i] >= ‘0’ && S[i] <= ‘9’ ;  ++i )

                n = 10 ∗ n  +  ( S[i] - ‘0’ ) ;

        return (n) ;
}
```

Eg : A C program which converts a string of digits into its numeric equivalent - version 2

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

void main( void )
{
        char  S[ ]  =  “12467” ;

        printf(“%ld \n”, atoi( S ) ) ;
}


long int atoi( char S[ ] )
{
        int  i = 0 , sign ;

        long int  n ;

        sign  =  ( S[i] == ‘-’ )  ?  -1  :  1 ;

        if( S[i] == ‘+’  ||  S[i] == ‘-’ )
```

```
                i++;


        for( n = 0 ;  isdigit( S[i] ) ;  ++i )

                n = 10 ∗ n  +  ( S[i] - ‘0’ ) ;

        return (sign ∗ n) ;
}
```

Eg : A C program which converts a string of digits into its numeric equivalent - version 2

```
# include < stdio.h >

void main( void )
{
        char  S[ ]  =  “12467” ;

        printf(“%lf \n”, atof( S ) ) ;
}


double atof( char S[ ] )
{
        double  val , power

        int  i , sign ;

        sign  =  ( S[i] == ‘-’ )  ?  -1  :  1 ;

        if( S[i] == ‘+’  ||  S[i] == ‘-’ )

                i++;

        for( val = 0.0 ;  isdigit( S[i] ) ;  ++i )

                val = 10.0 ∗ val  +  ( S[i] - ‘0’ ) ;

        if( S[i] == ‘.’ )

                i++;

        for( power = 1.0 ;  isdigit( S[i] ) ;  ++i )
        {
                val = 10.0 ∗ val  +  ( S[i] - ‘0’ ) ;

                power ∗= 10.0 ;
        }

        return (sign ∗ val / power) ;
}
```

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

```
# include < stdio.h >

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

        static int digits[10] ;

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

                        spaces++ ;

                else if ( isalpha( C ) )

                        letters++ ;

                else if ( isdigit( C ) )

                        digits[ C - ‘0’ ]++ ;

                else if ( ispunct( C ) )

                        puncts++ ;

                else

                        others++ ;

        printf(“%3d%3d%3d%3d\n”,spaces,letters,puncts,others);

        for( i = 0 ; i < 10 ; i++ )
```

```
                printf( "%3d", digits[i] );

        }
```

6.6 Arrays and Strings

While C supplies the data type *char*, it does not have a data type for character strings. Instead, you must represent a string as an array of characters. The array uses one cell for each character in the string, with the final cell holding the null character '\0'. When the compiler encounters a string constant such as

“Hello”

it is interpreted as five consecutive characters followed by a null character. We can store this constant in a character array of length 6.

```
                                                STR[0]  =  'H' ;
                                                STR[1]  =  'e'  ;
char  STR[6]  =  "Hello" ;          ⇒           STR[2]  =  'l'   ;
                                                STR[3]  =  'l'   ;
                                                STR[4]  =  'o'  ;
                                                STR[5]  =  '\0' ;
```

We cannot assign a string to an array directly.

i.e. STR = “Hello” ;

Eg. : Program to convert a positive integer to another base

```
# include < stdio.h >

void get_number_and_base( ) ;
void convert_number( ) ;
void display_converted_number( ) ;

int Converted_Number[20] ;
long int Number_to_Convert ;
int Base ;
int Index = 0 ;

void main( void )
{
        get_number_and_base( ) ;
        convert_number( ) ;
        display_converted_number( ) ;
}

void get_number_and_base( )
{
        printf(“ Enter number to be converted :  ” ) ;
        scanf(“%ld”, &Number_to_Convert ) ;

        printf(“ Enter the base : ” ) ;
        scanf(“%d”, &Base ) ;

        if( Base < 2  ||  Base > 16 )
        {
                printf(“Bad base - must be between 2 and 16 \n”) ;
                Base = 10 ;
        }
}

void convert_number( )
{
        do
        {
                Converted_Number[Index] = Number_to_Convert % Base;

                ++Index ;

                Number_to_Convert  / =  Base ;
        }
        while ( Number_to_Convert  !=  0 ) ;
}


void display_converted_number( )
{
        static char Base_digits[16]  =  { ‘0’ , ‘1’ , ‘2’ , ‘3’ , ‘4’ , ‘5’ , ‘6’ ,
                ‘7’ , ‘8’ , ‘9’ , ‘A’ , ‘B’ , ‘C’ , ‘D’ , ‘E’ , ‘F’ };

        int Next_digit ;
        printf(“ Converted number  =  ” ) ;

        for( --Index ;  Index > = 0 ;  --Index )
        {
                Next_digit  =  Converted_Number[Index] ;
                printf(“ %c” , Base_digits[Next_digit] ) ;
        }
}
```

Eg : The following C program inputs a line of text into an array ‘Line’. It returns the number of characters in the Line or EOF (-1) if end_of_file is detected.

```
# include < stdio.h >
# define MAXCHAR  132

char Line [MAXCHAR];

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

        while( ( C = getchar( ) ) != EOF )
        {
                if( C != ‘\n’ )

                        Line[Sub++] = C ;

                else
                        {

                                Line[Sub] = ‘\0’ ;

                                return (Sub) ;
                        }

                }

        return (EOF) ;

}
```

The statement

```
Line[Sub++] = C ;
```

assigns C to array element **Line[Sub]** and then increments Sub by 1.

Notice how we have replaced the newline character with the ‘**\0**’, this means that the array ‘**Line**’ contains a series of characters terminated by a character with code zero. This is the way we represent a string in C.

Eg : The following C program inputs a line of text into an array ‘Line’. It returns the number of characters in the Line or **EOF** (-1) if **end_of_file** is detected.

```
# include < stdio.h >
# define MAXCHAR  132

char Line [MAXCHAR];
int Getline( ) ;

void main( void )
{
        printf(“Type lines for input : \n”) ;

        while( Getline( ) != EOF )

                printf(“%s\n”, Line );

        printf(“End_of_text\n”);

}


int Getline( )
{
        int Sub = 0 , C ;

        while( ( C = getchar( ) ) != EOF )
        {
                if( C != ‘\n’ )

                        Line[Sub++] = C ;

                else
                        {

                                Line[Sub] = ‘\0’ ;

                                return (Sub) ;
                        }

                }

        return (EOF) ;

}
```

Notice that we have declared **Getline** as an integer function.

The statement

```
        printf("%s\n" , Line ) ;
```

outputs all characters in the array '**Line**' up to the zero character and then outputs the newline character.

The array '**Line**' is defined externally outside both routines (main and Getline) and it is global to both routines.

Instead, we could declare '**Line**' in main and pass it as an argument to Getline.

```
# include < stdio.h >
# define MAXCHAR  132

int Getline( ) ;

void main( void )
{
        char  Line [MAXCHAR] ;

        printf("Type lines for input : \n") ;

        while( Getline( Line ) != EOF )

                printf("%s\n", Line );

        printf("End_of_text\n");

}


int Getline( char Line[ ] )
{
        int Sub = 0 , C ;

        while( ( C = getchar( ) ) != EOF )
        {
                if( C != '\n' )

                        Line[Sub++] = C ;

                else
```

```
                        {

                                Line[Sub] = ‘\0’ ;

                                return (Sub) ;
                        }

                }

        return (EOF) ;

}
```

The next program copies text and tells us the length of the longest line.

```
# include < stdio.h >
# define MAXCHAR  132

int Getline( ) ;

void main( void )
{
        char  Line [MAXCHAR] ;
        int  length = 0 ,  maxlength = 0 ;

        printf(“Type lines for input : \n”) ;

        while( ( length = Getline( Line ) ) != EOF )
        {

                printf(“%s\n”, Line );

                if( length > maxlength )

                        maxlength = length ;

        }

        printf(“Length of the maximum line = %d \n ”, maxlength ) ;
        printf(“End_of_text\n” );

}


int Getline( char Line[ ] )
```

```
{
        int Sub = 0 , C ;

        while( ( C = getchar( ) ) != EOF )
        {
                if( C != ‘\n’ )

                        Line[Sub++] = C ;

                else
                        {

                                Line[Sub] = ‘\0’ ;

                                return (Sub) ;
                        }

                }

        return (EOF) ;

}
```

6.7 String Functions that use arrays

When standard input and output consider, we have to take account functions that use character arrays as function arguments. Specially, these functions are **gets**, **puts**, **fgets**, **fputs** and **sprintf** (which are string I/O functions), and **strcpy**, **strcat**, **strcmp**, and **strlen** (which are string manipulation functions).

strcpy, strcat, strlen, and strcmp

All of the functions discussed in this section are predefined in the *string.h* header file. Whenever you wish to use one of these functions, make certain that you include the header file in your program. The following program shows how to use the **strcpy** function :

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

# define LENGTH 17

void main( void )
```

```
{
        char  S1[LENGTH] = “Hello” ;
        char  S2[LENGTH] ;

        printf(“%s\n”, S1 ) ;

        strcpy( S2 , “World” ) ;
        printf(“%s\n”, S2 ) ;

        strcpy( S2 , S1 ) ;
        printf(“%s\n”, S2 ) ;

}
```

The **strcpy** function is used to copy the contents of one string, *S1*, into a second string, *S2*. The preceding program initializes *S1* with the message “Hello”. The first **strcpy** function call actually copies the “World” into *S2*. The second call to the **strcpy** function copies *S1* into the *S2* variable. The program outputs the following message :

```
Hello
World
Hello
```

You can use the **strcat** function to append two separate strings. Both strings must be null terminated and the result itself is null terminated. The following program builds on your understanding of the **strcpy** function and introduces **strcat** :

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

# define WORD_LENGTH  6
# define STRING_LENGTH  20

void main( void )
{
        char  P1[WORD_LENGTH] = “In” ;
        char  P2[WORD_LENGTH] = “ the ” ;
        char  P3[STRING_LENGTH] ;

        strcpy( P3 , P1 ) ;
        strcat( P3 ,  P2 ) ;
        strcat( P3 , “beginning...” ) ;
        printf(“%s\n”, P3 ) ;

}
```

In this example, both *P1* and *P2* are initialized, while *P3* is not. First, the program **strcpy** *P1* into *P3*. Next, the **strcat** function is used to concatenate *P2* (“ the “) to “In”, which is stored in *P3*. The last **strcat** function call demonstrates how a string constant can be concatenated to a string. Here, “beginning...” is concatenated to the now current contents of *P3* (“In the ”). The program outputs

```
In the beginning...
```

The following C program demonstrates how to use the **strcmp** function :

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

void main( void )
{
        char String1[ ]  =  “one” ;
        char String2[ ]  =  “one” ;
        int  result = 0 ;

        if ( strlen( String2 ) >=  strlen( String1 )

                result = strcmp( String1 , String2 ) ;

        printf(“The string %s found”, result == 0 ?  “was” :  “wasn’t” ) ;

}
```

The **strlen** function returns the integer length of the string pointed to. In the preceding program, it is used in two different forms just to show what it can do. The first call to the function is actually encountered within the if condition. Remember, all test conditions must evaluate to a TRUE (!0) or FALSE (0). The if test takes the results returned from the two calls to **strcmp** and then asks the relational question greater than or equal to (>=). If the length of *String2* is greater than or equal to that of *String1*, the **strcmp** function is invoked.

You might wonder why you use the greater than or equal to test instead of an equal to test. This method illustrates further how **strcmp** works. The **strcmp** function begins comparing two strings starting with the first character in each string. If both strings are identical, the function returns a value of 0. However, if the two strings aren’t identical, the function returns a value less than 0 if *String1* is less than *String2*, or a value greater than 0 if *String1* is greater than *String2*. The relational test (>=) was used in case you would want to modify the code to include a report of equality, greater than, or less than for the compared strings.

The program terminates by using the value returned by *result*, along with the conditional operator (**?:**) to determine which message is printed. For this example, the program output is as follows :

```
The string was found
```

gets, puts, fgets, fputs, and sprintf

Strings can be output with *printf* (using the %s descriptor). Another function **puts** outputs a string appending a newline('\n').

```
char  Msg[]  =  "Hello World" ;

puts(Msg) ;
```

would output "Hello World" and move the cursor onto a new line.

The opposite of **puts** is **gets** which inputs a line from the terminal replacing the newline character with a null character.

Eg : A C program demonstrating string I/O functions

```
# include < stdio.h >

# define SIZE 20

void main( void )
{
        char test_array[SIZE] ;

        fputs("Please enter the first string  :   " , stdout ) ;
        gets(test_array) ;
        fputs("The first string entered is    :   " , stdout ) ;
        puts(test_array) ;

        fputs("Please enter the second string  :  " , stdout ) ;
        fgets(test_array , SIZE , stdin ) ;
        fputs("The second string entered is    :   " , stdout ) ;
        fputs(test_array , stdout ) ;

        sprintf(test_array , "This was %s a test" , "just" ) ;
        fputs("sprintf( ) created                     :   " , stdout ) ;
        fputs(test_array , stdout ) ;

}
```

Here is the output from the first run of the program :

```
Please enter the first string1          :       string one
The first string entered is             :       string one
Please enter the second string          :       string two
The second string entered is            :       string two
sprintf( ) created                      :       This was just a test
```

Take care when running the program. The **gets** function receives characters from standard input(**stdin**, the keyboard by default for most computers) and places them into the array whose name is passed to the function. When you press ENTER to terminate the string, a newline character is transmitted. When the **gets** function receives this newline character, it changes it into a null character, ensuring that the character array contains a string. No checking occurs to ensure that the array is large enough to hold all the characters entered.

The **puts** function echoes to the terminal just what was entered with **gets**. It also appends a newline character to the string where the null character appeared. Remember, the null character was automatically inserted into the string by the **gets** function. Therefore, strings that are properly entered with **gets** can be displayed with **puts**.

When you use the **fgets** function, you can guarantee a maximum number of input characters. This function stops reading the designated file stream when one fewer characters are read than the second argument specifies. Since the *test_array size* is 20, only 19 characters will be read by **fgets** from **stdin**. A null character is automatically placed into the string in the twentieth position, and if you entered a newline character from the keyboard it would be retained in the string (it would appear before the null). The **fgets** function does not eliminate the newline character, as **gets** did, but merely appends the null character so that a valid string is stored. Much like **gets** and **puts**, **fgets** and **fputs** are symmetrical. **fgets** does not eliminate the newline character, nor does **fputs** add one.

The function **sprintf**, which stands for “string **printf**,” uses a control string with conversion characters, just like **printf**. However, **sprintf** places the resulting formatted data in a string rather than immediately sending the result to standard output. This can be beneficial if the exact same output must be created twice-for example, when the same string must be output to both the display monitor and the printer.

To review these functions :

- gets converts newline to a null
- puts converts null to a newline
- fgets retains newline and appends a null

fputs drops the null and does not add a newline; instead it uses the retained newline (if one was entered).