UNIT-IV

C Preprocessor

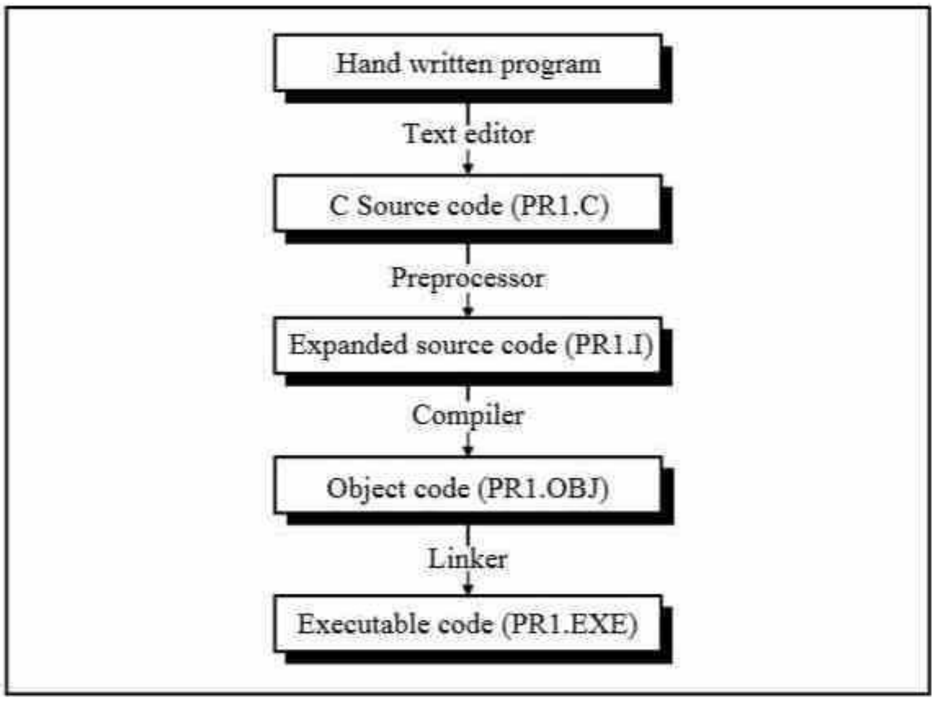
The C preprocessor is exactly what its name implies. It is aprogram that processes our source program before it is passed to the compiler.

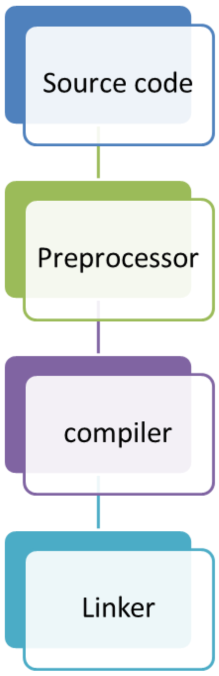
Features of C Preprocessor

There are several steps involved from the stage of writing a C program to the stage of getting it executed.

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before thefirst function definition. We would learn the following preprocessor directives here:

(a) Macro expansion
(b) File inclusion
(c) Conditional Compilation
(d) Miscellaneous directives





Macro Expansion

Macro represents a group of commonly used statements in the source programming language. Macro Processor replaces each macro instruction with the corresponding group of source language statements. This is known as the expansion of macros.

You can define a macro in C using the #define preprocessor directive.

Have a look at the following program.

```
#define UPPER 25
main( )

{

    int  i ;

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

        printf ( "\n%d", i ) ;

}
```

In this program instead of writing 25 in the **for** loop we are writingit in the form of UPPER, which has already been defined before **main( )** through the statement,

```
#define UPPER 25
```

This statement is called ‘macro definition’ or more commonly, justa ‘macro’.

Macros with Arguments

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

Function-like macros can take arguments, just like true functions. To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like. The parameters must be valid C identifiers, separated by commas and optionally whitespace.

```
#define AREA(x) ( 3.14 * x * x )
```

Macros versus Functions

In a macro call the preprocessor replaces the macro template with its macro expansion, in a stupid, unthinking, literal way. As against this, in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

File Inclusion

The second preprocessor directive we'll explore in this chapter is file inclusion. This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

```
#include "filename"
```

and it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program.

(a) If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are **#included** at the beginning of main program file.

(b) There are some functions and some macro definitions that we need almost in all programs that we write.

Actually there exist two ways to write **#include** statement. These are:

```
#include
"filename"
#include
<filename>
```

Conditional Compilation

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**, which have the general form:

```
#ifdef macroname

    statement 1 ;

    statement 2 ;

    statement 3 ;

    #endif
```

If **macroname** has been **#define**d, the block of code will be processed as usual; otherwise not.

Therefore the solution is to use conditional compilation as shown below.

```
main( )

{

    #ifdef OKAY

        statement 1 ;
```

```
            statement 2 ; /* detects virus */
            statement 3 ;

            statement 4 ; /* specific to stone virus */#endif

        statement 5 ;

        statement 6 ;

        statement 7 ;

    }
```

Here, statements 1, 2, 3 and 4 would get compiled only if the macro OKAY has been defined, and we have purposefully omitted the definition of the macro OKAY. At a later date, if we want that these statements should also get compiled all that we are required to do is to delete the **#ifdef** and **#endif** statements.

#if **and** ***#elif*** **Directives**

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expressionis nonzero, then subsequent lines upto a **#else**, **#elif** or **#endif** are compiled, otherwise they are skipped.

A simple example of **#if** directive is shown below:

```
main( )

{

    #if TEST <= 5

        statement 1 ;

        statement 2 ;

        statement 3 ;#else

        statement 4 ;

        statement 5 ;

        statement 6 ;#endif

}
```

Arrays

What are Arrays

Suppose we wish to arrange the percentage marks obtained by 100 students in ascending order. In such a case we have two options to store these marks in memory:

(a) Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.

(b) Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Thus, an array is a collection of similar elements. These similar elements could be all **int**s, or all **float**s, or all **char**s, etc. Usually, the array of characters is called a 'string'.

Ex:

```
int marks[30] ;
```

Accessing Elements of an Array

The number inthe brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Thus, **marks[2]** is not the second element of the array, but the third.

Array Declaration

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want. In our program we have done this with the statement:

```
for ( i = 0 ; i <= 29 ; i++ )
{
    printf ( "\nEnter marks " ) ;
    scanf ( "%d", &marks[i] ) ;
}
```

To fix our ideas, let us revise whatever we have learnt aboutarrays:

(a) An array is a collection of similar elements.
(b) The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
(c) An array is also known as a subscripted variable.
(d) Before using an array its type and dimension must be declared.
(e) However big an array its elements are always stored in contiguous memory locations. This is a very important point which we would discuss in more detail later on.

More on Arrays

Array is a very popular data type with C programmers.

Array Initialisation

Ex;

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;

int n[ ] = { 2, 4, 12, 5, 45, 5 } ;

float  press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;
```

Note the following points carefully:

(a) Till the array elements are not given any specific values, they are supposed to contain garbage values.
(b) If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

Array Elements in Memory

Consider the following array declaration:

```
int arr[8] ;
```

What happens in memory when we make this declaration? 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers.

Bounds Checking

In C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself.

Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In the call by value we pass values of array elements to the function, whereas in the call by reference we pass addresses of array elements to the function.

```
/* Demonstration of call by value */
main( )
{
int  i ;
int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

for ( i = 0 ; i <= 6 ; i++ )
display ( marks[i] ) ;
}

display ( int m )
```

```
/* Demonstration of call by reference *
/main( )
{
int  i ;
int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

for ( i = 0 ; i <= 6 ; i++ )
disp ( &marks[i] ) ;
}

disp ( int *n )
{
```

{
printf ("%d ", m) ;
}
And here’s the output...
 	
55 65 75 56 78 78 90
	printf ("%d ", *n) ;
	}
	And here’s the output...
	
	55 65 75 56 78 78 90

Pointers and Arrays

(a) Addition of a number to a pointer. For example,

```
int  i = 4, *j, *k ;
j = &i ;

j = j + 1 ;j
= j + 9 ;k
= j + 3 ;
```

Now we will try to correlate the following two facts, which we have learnt above:

(a) Array elements are always stored in contiguous memory locations.
(b) A pointer when incremented always points to an immediately next location of its type.

Passing an Entire Array to a Function

```
/* Demonstration of passing an entire array to a function */main( )

{

    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;

    dislpay ( &num[0], 6 ) ;

}


display ( int *j, int  n )

{

int  i ;


    for ( i = 0 ; i <= n - 1 ; i++ )

    {

        printf ( "\nelement = %d", *j ) ;

        j++ ; /* increment pointer to point to next element */

    }
```

}

Two Dimensional Arrays

A two-dimensional array in C can be thought of as **a matrix with rows and columns**. The general syntax used to declare a two-dimensional array is: A two-dimensional array is an array of several one-dimensional arrays. Following is an array with five rows, each row has three columns:

```
int my_array[4][2];
```

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

Initialising a 2-Dimensional Array

Syntax:

data_type array_name[rows][columns];

ex:

```
int stud[4][2];
```

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

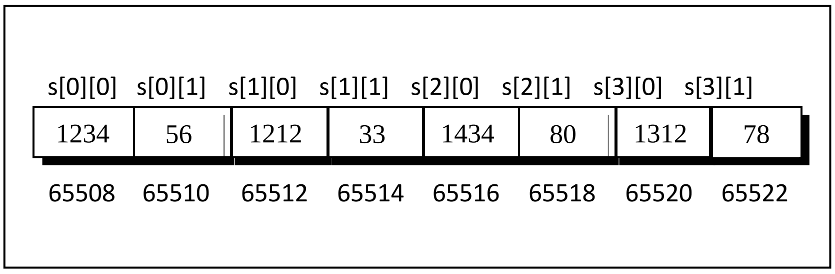
EX:

```
int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;

int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;
```

RESULT:

```
12      34      23

45      56      45
```

Memory Map of a 2-Dimensional Array



Let us reiterate the arrangement of array elements in a two-dimensional array of students, which contains roll nos. in one column and the marks in the other.

Pointer to an Array

If we can have a pointer to an integer, a pointer to a float, a pointerto a char, then can we not have a pointer to an array.

```
/* Usage of pointer to an array */main(
)

{

    int s[5][2] = {
                        { 1234, 56 },

                        { 1212, 33 },

                        { 1434, 80 },

                  }     { 1312, 78 }
                  ;

    int ( *p )[2] ;
    int  i, j, *pint ;
    for ( i = 0 ; i <= 3 ; i++ )

    {

        p = &s[i] ;
        pint = p ;
        printf ( "\n" ) ;

        for ( j = 0 ; j <= 1 ; j++ )

            printf ( "%d ", *( pint + j ) ) ;

    }

}
```

And here is the output...

```
    1234  56
    1212  33
    1434  80
    1312  78
```

Passing 2-D Array to a Function

There are three ways in which we can pass a 2-D array to afunction. These are illustrated in the following program.

```
/* Three ways of accessing a 2-D array */main(

)

{

     int a[3][4] =   {
                         1, 2, 3, 4,

                         5, 6, 7, 8,

                         9, 0, 1, 6
                     } ;


    clrscr( ) ;

    display ( a, 3, 4 ) ;

    show ( a, 3, 4 ) ;

    print ( a, 3, 4 ) ;

}


display ( int *q, int  row, int  col )

{

    int  i, j ;


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

```
        {
                for ( j = 0 ; j < col ; j++ )
                        printf ( "%d ", * ( q + i * col + j ) ) ;


                printf ( "\n" ) ;
        }
        printf ("\n" ) ;
}

show ( int ( *q )[4], int  row, int  col )
{
        int i, j ;
        int *p ;

        for ( i = 0 ; i < row ; i++ )
        {
                p = q + i ;
                for ( j = 0 ; j < col ; j++ )
                        printf ( "%d ", * ( p + j ) ) ;

                printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
}

print ( int q[ ][4], int  row, int  col )
{
        int  i, j ;

        for ( i = 0 ; i < row ; i++ )
        {
                for ( j = 0 ; j < col ; j++ )
                        printf ( "%d ", q[i][j] ) ;

                printf ( "\n" ) ;
        }
        printf ( "\n" ) ;
}
```

And here is the output…

```
1 2 3 4
5 6 7 8
9 0 1 6

1 2 3 4
5 6 7 8
9 0 1 6

1 2 3 4
5 6 7 8
9 0 1 6
```

Three-Dimensional Array

```
int arr[3][4][2] = {
                      {
                          { 2, 4 },
                          { 7, 8 },
                          { 3, 4 },
                          { 5, 6 }
                      }
                      ,
                      {
                          { 7, 6 },
                          { 3, 4 },
                          { 5, 3 },
                          { 2, 3 }
                      }
                      ,
                      {
                          { 8, 9 },
                          { 7, 2 },
                          { 3, 4 },
                          { 5, 1 },
                      }
                  } ;
```

A three-dimensional array can be thought of as an array of arrays of arrays.

would possibly help you in visualising the situation better.

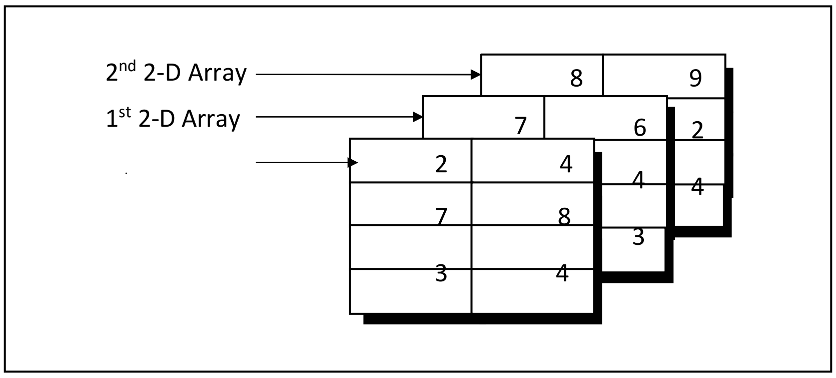


Figure 8.9

Again remember that the arrangement shown above is only conceptually true. In memory the same array elements are stored linearly as shown in Figure 8.10.

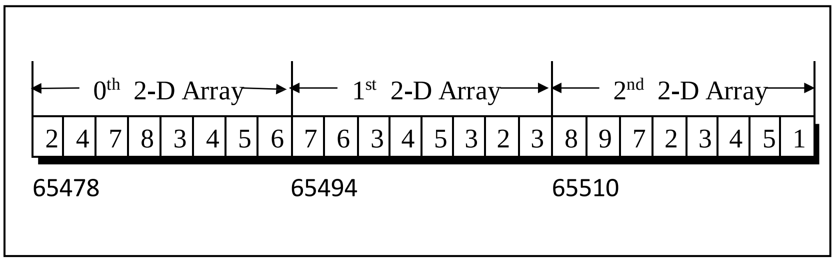


Figure 8.10