(j) If a global variable is to be defined, then the extern keyword is necessary in its declaration.

(k) The address of register variable is not accessible.

[D] Following program calculates the sum of digits of the number 12345. Go through it and find out why is it necessary to declare the storage class of the variable **sum** as **static**.

```
main( )
{
    int a ;
    a = sumdig ( 12345 ) ;
    printf ( "\n%d", a ) ;
}
sumdig ( int num )
{
    static int sum ;
    int a, b ;
    a = num % 10 ;
    b = ( num - a ) / 10 ;
    sum = sum + a ;
    if ( b != 0 )
        sumdig ( b ) ;
    else
        return ( sum ) ;
}
```

7 The C Preprocessor

- Features of C Preprocessor
- Macro Expansion
  - Macros with Arguments
  - Macros versus Functions
- File Inclusion
- Conditional Compilation
- *#if* and *#elif* Directives
- Miscellaneous Directives
  - *#undef* Directive
  - *#pragma* Directive
- The Build Process
  - Preprocessing
  - Compilation
  - Assembling
  - Linking
- Summary
- Exercise

The C preprocessor is exactly what its name implies. It is a program that processes our source program before it is passed to the compiler. Preprocessor commands (often known as directives) form what can almost be considered a language within C language. We can certainly write C programs without knowing anything about the preprocessor or its facilities. But preprocessor is such a great convenience that virtually all C programmers rely on it. This chapter explores the preprocessor directives and discusses the pros and cons of using them in programs.

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 combination of these steps is known as the 'Build Process'. The detailed build process is discussed in the last section of this chapter. At this stage it would be sufficient to note that before a C program is compiled it is passed through another program called 'Preprocessor'. The C program is often known as 'Source Code'. The Preprocessor works on the source code and creates 'Expanded Source Code'. If the source code is stored in a file PR1.C, then the expanded source code gets stored in a file PR1.I. It is this expanded source code that is sent to the compiler for compilation.

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

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

Let us understand these features of preprocessor one by one.

Macro Expansion

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 writing it 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, just a 'macro'. What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25. Here is another example of macro definition.

```
#define PI  3.1415
main( )
{
    float  r = 6.25 ;
    float  area ;

    area = PI * r * r ;
    printf ( "\nArea of circle = %f", area ) ;
}
```

UPPER and PI in the above programs are often called 'macro templates', whereas, 25 and 3.1415 are called their corresponding 'macro expansions'.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

In C programming it is customary to use capital letters for macro template. This makes it easy for programmers to pick out all the macro templates when reading through the program.

Note that a macro template and its macro expansion are separated by blanks or tabs. A space between # and **define** is optional. Remember that a macro definition is never to be terminated by a semicolon.

And now a million dollar question... why use **#define** in the above programs? What have we gained by substituting PI for 3.1415 in our program? Probably, we have made the program easier to read. Even though 3.1415 is such a common constant that it is easily recognizable, there are many instances where a constant doesn't reveal its purpose so readily. For example, if the phrase "\x1B[2J" causes the screen to clear. But which would you find easier to understand in the middle of your program "\x1B[2J" or "CLEARSCREEN"? Thus, we would use the macro definition

```
#define CLEARSCREEN "\x1B[2J"
```

Then wherever CLEARSCREEN appears in the program it would automatically be replaced by "\x1B[2J" before compilation begins.

There is perhaps a more important reason for using macro definition than mere readability. Suppose a constant like 3.1415 appears many times in your program. This value may have to be changed some day to 3.141592. Ordinarily, you would need to go through the program and manually change each occurrence of the constant. However, if you have defined PI in a **#define** directive, you only need to make one change, in the **#define** directive itself:

```
#define PI 3.141592
```

Beyond this the change will be made automatically to all occurrences of PI before the beginning of compilation.

In short, it is nice to know that you would be able to change values of a constant at all the places in the program by just making a change in the **#define** directive. This convenience may not matter for small programs shown above, but with large programs macro definitions are almost indispensable.

But the same purpose could have been served had we used a variable **pi** instead of a macro template **PI**. A variable could also have provided a meaningful name for a constant and permitted one change to effect many occurrences of the constant. It's true that a variable can be used in this way. Then, why not use it? For three reasons it's a bad idea.

Firstly, it is inefficient, since the compiler can generate faster and more compact code for constants than it can for variables. Secondly, using a variable for what is really a constant encourages sloppy thinking and makes the program more difficult to understand: if something never changes, it is hard to imagine it as a variable. And thirdly, there is always a danger that the variable may inadvertently get altered somewhere in the program. So it's no longer a constant that you think it is.

Thus, using **#define** can produce more efficient and more easily understandable programs. This directive is used extensively by C programmers, as you will see in many programs in this book.

Following three examples show places where a **#define** directive is popularly used by C programmers.

A **#define** directive is many a times used to define operators as shown below.

```
#define AND &&
#define OR ||
main( )
{
    int f = 1, x = 4, y = 90 ;

    if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
        printf ( "\nYour PC will always work fine..." ) ;
    else
        printf ( "\nIn front of the maintenance man" ) ;
}
```

A **#define** directive could be used even to replace a condition, as shown below.

```
#define AND &&
#define ARANGE ( a > 25 AND a < 50 )
main( )
{
    int a = 30 ;

    if ( ARANGE )
        printf ( "within range" ) ;
    else
        printf ( "out of range" ) ;
}
```


A **#define** directive could be used to replace even an entire C statement. This is shown below.

```
#define FOUND printf ( "The Yankee Doodle Virus" ) ;
main( )
{
    char signature ;

    if ( signature == 'Y' )
        FOUND
    else
        printf ( "Safe... as yet !" ) ;
}
```

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.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
    float r1 = 6.25, r2 = 2.5, a ;

    a = AREA ( r1 ) ;
    printf ( "\nArea of circle = %f", a ) ;
    a = AREA ( r2 ) ;
    printf ( "\nArea of circle = %f", a ) ;
}
```

Here's the output of the program...

```
Area of circle = 122.656250
Area of circle = 19.625000
```

In this program wherever the preprocessor finds the phrase **AREA(x)** it expands it into the statement **(3.14 * x * x)**. However, that's not all that it does. The **x** in the macro template **AREA(x)** is an argument that matches the **x** in the macro expansion **(3.14 * x * x)**. The statement **AREA(r1)** in the program causes the variable **r1** to be substituted for **x**. Thus the statement **AREA(r1)** is equivalent to:

```
( 3.14 * r1 * r1 )
```

After the above source code has passed through the preprocessor, what the compiler gets to work on will be this:

```
main( )
{
    float r1 = 6.25, r2 = 2.5, a ;

    a = 3.14 * r1 *r1 ;
    printf ( "Area of circle = %f\n", a ) ;
    a = 3.14 *r2 * r2 ;
    printf ( "Area of circle = %f", a ) ;
}
```

Here is another example of macros with arguments:

```
#define ISDIGIT(y) ( y >= 48 && y <= 57 )
main( )
{
    char ch ;

    printf ( "Enter any digit " ) ;
    scanf ( "%c", &ch ) ;

    if ( ISDIGIT ( ch ) )
        printf ( "\nYou entered a digit" ) ;
    else
        printf ( "\nIllegal input" ) ;
}
```


Here are some important points to remember while writing macros with arguments:

(a) Be careful not to leave a blank between the macro template and its argument while defining the macro. For example, there should be no blank between **AREA** and **(x)** in the definition,

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

If we were to write **AREA (x)** instead of **AREA(x)**, the **(x)** would become a part of macro expansion, which we certainly don't want. What would happen is, the template would be expanded to

```
( r1 ) ( 3.14 * r1 * r1 )
```

which won't run. Not at all what we wanted.

(b) The entire macro expansion should be enclosed within parentheses. Here is an example of what would happen if we fail to enclose the macro expansion within parentheses.

```
#define SQUARE(n) n * n
main( )
{
    int j ;

    j = 64 / SQUARE ( 4 ) ;
    printf ( "j = %d", j ) ;
}
```

The output of the above program would be:

```
j = 64
```

whereas, what we expected was j = 4.

What went wrong? The macro was expanded into

```
j = 64 / 4 * 4 ;
```

which yielded 64.

(c) Macros can be split into multiple lines, with a '\' (back slash) present at the end of each line. Following program shows how we can define and use multiple line macros.

```
#define HLINE   for ( i = 0 ; i < 79 ; i++ ) \
                    printf ( "%c", 196 ) ;

#define VLINE( X, Y ) {\
                        gotoxy ( X, Y ) ; \
                        printf ( "%c", 179 ) ; \
                      }
main( )
{
   int i, y ;
   clrscr( ) ;

   gotoxy ( 1, 12 ) ;
   HLINE

   for ( y = 1 ; y < 25 ; y++ )
      VLINE ( 39, y ) ;
}
```

This program draws a vertical and a horizontal line in the center of the screen.

(d) If for any reason you are unable to debug a macro then you should view the expanded code of the program to see how the macros are getting expanded. If your source code is present in the file PR1.C then the expanded source code would be stored

in PR1.I. You need to generate this file at the command prompt by saying:

```
cpp pr1.c
```

Here CPP stands for C PreProcessor. It generates the expanded source code and stores it in a file called PR1.I. You can now open this file and see the expanded source code. Note that the file PR1.I gets generated in C:\TC\BIN directory. The procedure for generating expanded source code for compilers other than Turbo C/C++ might be a little different.

Macros versus Functions

In the above example a macro was used to calculate the area of the circle. As we know, even a function can be written to calculate the area of the circle. Though macro calls are 'like' function calls, they are not really the same things. Then what is the difference between the two?

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.

This brings us to a question: when is it best to use macros with arguments and when is it better to use a function? Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.

If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a function is used, then even if it is called from hundred different places in

the program, it would take the same amount of space in the program.

But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.

Moral of the story is—if the macro is simple and sweet like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a 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. Of course this presumes that the file being included is existing. When and why this feature is used? It can be used in two cases:

(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. These commonly

needed functions and macro definitions can be stored in a file, and that file can be included in every program we write, which would add all the statements in this file to our program as if we have typed them in.

It is common for the files that are to be included to have a .h extension. This extension stands for 'header file', possibly because it contains statements which when included go to the head of your program. The prototypes of all the library functions are grouped into different categories and then stored in different header files. For example prototypes of all mathematics related functions are stored in the header file 'math.h', prototypes of console input/output functions are stored in the header file 'conio.h', and so on.

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

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

The meaning of each of these forms is given below:

`#include "goto.h"`	This command would look for the file **goto.h** in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.
`#include <goto.h>`	This command would look for the file **goto.h** in the specified list of directories only.

The include search path is nothing but a list of directories that would be searched for the file being included. Different C compilers let you set the search path in different manners. If you are using Turbo C/C++ compiler then the search path can be set up by selecting 'Directories' from the 'Options' menu. On doing this

a dialog box appears. In this dialog box against 'Includ Directories' we can specify the search path. We can also specif multiple include paths separated by ';' (semicolon) as show below:

```
c:\tc\lib ; c:\mylib ; d:\libfiles
```

The path can contain maximum of 127 characters. Both relative and absolute paths are valid. For example '..\dir\incfiles' is a valid path.

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 **#defined**, the block of code will be processed as usual; otherwise not.

Where would **#ifdef** be useful? When would you like to compile only a part of your program? In three cases:

(a) To "comment out" obsolete lines of code. It often happens that a program is changed at the last minute to satisfy a client. This involves rewriting some part of source code to the client's satisfaction and deleting the old code. But veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was

Now you would definitely not like to retype the deleted code again.

One solution in such a situation is to put the old code within a pair of /* */ combination. But we might have already written a comment in the code that we are about to "comment out". This would mean we end up with nested comments. Obviously, this solution won't work since we can't nest comments in C.

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.

(b) A more sophisticated use of **#ifdef** has to do with making the programs portable, i.e. to make them work on two totally different computers. Suppose an organization has two

different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with **#ifdef**. For example:

```
main( )
{
    #ifdef INTEL
        code suitable for a Intel PC
    #else
        code suitable for a Motorola PC
    #endif
    code common to both the computers
}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else** control instruction of C.

If you want to run your program on a Motorola PC, just add a statement at the top saying,

```
#define INTEL
```

Sometimes, instead of **#ifdef** the **#ifndef** directive is used. The **#ifndef** (which means 'if not defined') works exactly opposite to **#ifdef**. The above example if written using **#ifndef**, would look like this:

```
main( )
{
    #ifndef INTEL
        code suitable for a Intel PC
    #else
        code suitable for a Motorola PC
    #endif
    code common to both the computers
}
```

(c) Suppose a function **myfunc( )** is defined in a file 'myfile.h' which is **#included** in a file 'myfile1.h'. Now in your program file if you **#include** both 'myfile.h' and 'myfile1.h' the compiler flashes an error 'Multiple declaration for **myfunc**'. This is because the same file 'myfile.h' gets included twice. To avoid this we can write following code in the header file.

```
/* myfile.h */
#ifndef __myfile_h
    #define __myfile_h

    myfunc( )
    {
        /* some code */
    }

#endif
```

First time the file 'myfile.h' gets included the preprocessor checks whether a macro called **__myfile_h** has been defined or not. If it has not been then it gets defined and the rest of the code gets included. Next time we attempt to include the same file, the inclusion is prevented since **__myfile_h** already stands defined. Note that there is nothing special about **__myfile_h**. In its place we can use any other macro as well.

#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 expression is 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
}
```

If the expression, **TEST <= 5** evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled. In place of the expression **TEST <= 5** other expressions like **(LEVEL == HIGH || LEVEL == LOW)** or **ADAPTER == CGA** can also be used.

If we so desire we can have nested conditional compilation directives. An example that uses such directives is shown below.

```
#if ADAPTER == VGA
    code for video graphics array
#else
    #if ADAPTER == SVGA
        code for super video graphics array
    #else
        code for extended graphics adapter
    #endif
#endif
```

The above program segment can be made more compact by using another conditional compilation directive called **#elif**. The same program using this directive can be rewritten as shown below.

Observe that by using the **#elif** directives the number of **#endifs** used in the program get reduced.

```
#if ADAPTER == VGA
    code for video graphics array
#elif ADAPTER == SVGA
    code for super video graphics array
#else
    code for extended graphics adapter
#endif
```

Miscellaneous Directives

There are two more preprocessor directives available, though they are not very commonly used. They are:

(a) #undef
(b) #pragma

#undef Directive

On some occasions it may be desirable to cause a defined name to become 'undefined'. This can be accomplished by means of the **#undef** directive. In order to undefine a macro that has been earlier **#defined**, the directive,

```
#undef macro template
```

can be used. Thus the statement,

```
#undef PENTIUM
```

would cause the definition of PENTIUM to be removed from the system. All subsequent **#ifdef PENTIUM** statements would evaluate to false. In practice seldom are you required to undefine a macro, but for some reason if you are required to, then you know that there is something to fall back upon.

#pragma Directive

Imp

This directive is another special-purpose directive that you can use to turn on or off certain features. Pragmas vary from one compiler to another. There are certain pragmas available with Microsoft C compiler that deal with formatting source listings and placing comments in the object file generated by the compiler. Turbo C/C++ compiler has got a pragma that allows you to suppress warnings generated by the compiler. Some of these pragmas are discussed below.

(a) **#pragma startup** and **#pragma exit**: These directives allow us to specify functions that are called upon program startup (before **main( )**) or program exit (just before the program terminates). Their usage is as follows:

```
void fun1( ) ;
void fun2( ) ;

#pragma startup fun1
#pragma exit fun2

main( )
{
    printf ( "\nInside maim" ) ;
}

void fun1( )
{
    printf ( "\nInside fun1" ) ;
}

void fun2( )
{
    printf ( "\nInside fun2" ) ;
}
```


And here is the output of the program.

```
Inside fun1
Inside main
Inside fun2
```

Note that the functions **fun1( )** and **fun2( )** should neither receive nor return any value. If we want two functions to get executed at startup then their pragmas should be defined in the reverse order in which you want to get them called.

(b) **#pragma warn**: On compilation the compiler reports Errors and Warnings in the program, if any. Errors provide the programmer with no options, apart from correcting them. Warnings, on the other hand, offer the programmer a hint or suggestion that something may be *wrong* with a particular piece of code. Two most common situations when warnings are displayed are as under:

- If you have written code that the compiler's designers (or the ANSI-C specification) consider bad C programming practice. For example, if a function does not return a value then it should be declared as **void**.
- If you have written code that might cause run-time errors, such as assigning a value to an uninitialised pointer.

The **#pragma warn** directive tells the compiler whether or not we want to suppress a specific warning. Usage of this pragma is shown below.

```
#pragma warn –rvl   /* return value */
#pragma warn –par   /* parameter not used */
#pragma warn –rch   /* unreachable code */

int f1( )
{
   int a = 5 ;
}

void f2 ( int x )
{
   printf ( "\ninside f2" ) ;
}

int f3( )
{
   int x = 6 ;
   return x ;
   x++ ;
}

void main( )
{
   f1( ) ;
   f2 ( 7 ) ;
   f3( ) ;
}
```

If you go through the program you can notice three problems immediately. These are:

(a) Though promised, **f1( )** doesn't return a value.

(b) The parameter **x** that is passed to **f2( )** is not being used anywhere in **f2( )**.

(c) The control can never reach **x++** in **f3( )**.

If we compile the program we should expect warnings indicating the above problems. However, this does not happen since we have suppressed the warnings using the **#pragma** directives. If we replace the '–' sign with a '+' then these warnings would be flashed on compilation. Though it is a bad practice to suppress warnings, at times it becomes useful to suppress them. For example, if you have written a huge program and are trying to compile it, then to begin with you are more interested in locating the errors, rather than the warnings. At such times you may suppress the warnings. Once you have located all errors, then you may turn on the warnings and sort them out.

The Build Process

There are many steps involved in converting a C program into an executable form. Figure 7.1 shows these different steps along with the files created during each stage.

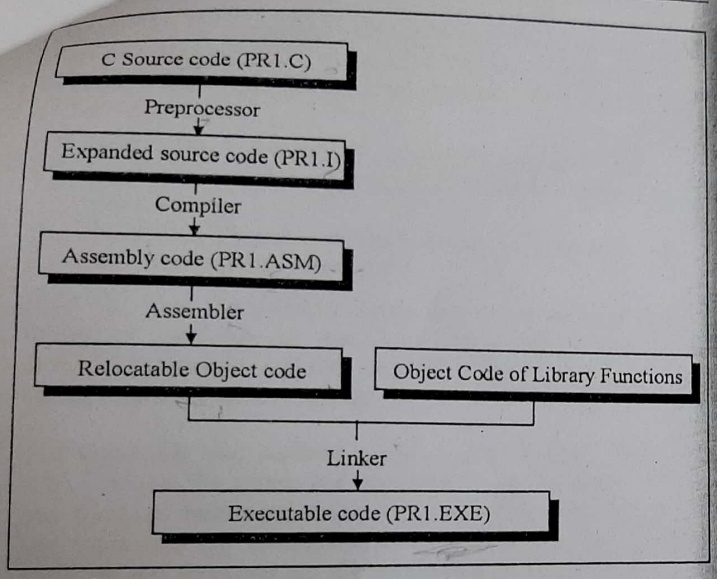

Figure 7.1

Many software development tools like TC++ and VC++ hide many of the steps shown in Figure 7.1 from us. However, it is important to understand these steps for two reasons:

(a) It would help you understand the process much better than just believing that some kind of 'magic' generates the executable code.
(b) If you are to alter any of these steps then you would know how to do it.

Let us now understand the steps mentioned in Figure 7.1 in detail.

Preprocessing

During this step the C source code is expanded based on the preprocessor directives like **#define**, **#include**, **#ifdef**, etc. The expanded source code is stored in an intermediate file with .i extension. Thus if our source code is stored in PR1.C then the expanded source code is stored in PR1.I. The expanded source code is also in C.

Compilation

The expanded source code is then passed to the compiler where syntax errors are identified. These errors are displayed along with warnings, if any. As we saw in the last section using the **#pragma** directive we can control which warnings are to be displayed / hidden.

If the expanded source code is error free then compiler translates the expanded source code in C into an equivalent assembly language program. Different processors (CPUs) support different set of assembly instructions using which they can be programmed. Hence the compiler targeted for Intel Pentium III platform would generate the assembly code using the instructions understood by Intel Pentium III. Likewise, the same C program when compiled using a compiler targeted for Intel Pentium IV platform is likely to generate a different assembly language code. The assembly code is typically stored in .ASM file. Thus for PR1.I the assembly code would be stored in PR1.ASM.

Assembling

The job of the Assembler is to translate .ASM program into Relocatable Object code. Thus assembler would convert PR1.ASM into PR1.OBJ. The .OBJ file is a specially formatted binary file that contains the set of machine language instructions and data resulting from the language translation process. Although parts of

this file contain executable code, the object file is not intended to be executed directly. The object file contains a header and several sections. The header describes the sections that follow it. These sections are:

(a) Text section – This section contains one more code blocks.
(b) Data Section – This section contains global variables and their initial values.
(c) Bss (Block Started by Symbol) section – This section contains uninitialized global variables.

Here the word 'Relocatable' means that the program is complete except for one thing—no specific memory addresses have yet been assigned to the code and data sections in the relocatable code. All the addresses are relative offsets.

The object file also contains a table called 'Symbol Table'. This table contains the names and locations of all the global variables and functions referenced within the source file. Note that parts of this table may be incomplete, however, because not all of the variables and functions are always defined in the same file. These are the symbols that refer to variables and functions defined in other source files. It is up to the linker to resolve such unresolved references.

Linking

For multiple source files multiple object files would get created. As said earlier, the object files themselves are individually incomplete, because some variable and function references may have not yet been resolved. The job of the 'Linker' is to combine these individual object files with object files containing library functions, and, in the process, resolve all of the unresolved symbols. The output of the linker is an executable file in machine language. This file contains all of the code and data from the input object files and is in the same object file format.

The linker merges the Text, Data, and Bss sections of the object files submitted to it. So, when the linker is finished executing, all of the machine language code from all of the input object files will be in the Text section of the executable file, and all of the initialized and uninitialized variables will reside in the new Data and Bss sections, respectively.

While the linker is in the process of merging the section contents, it is also on the lookout for unresolved symbols. For example, if one object file contains an unresolved reference to a variable named **a** and a variable with that same name is declared in one of the other object files, the linker will match them up. The unresolved reference will be replaced with a reference to the actual variable. In other words, if **a** is located at offset 14 of the Data section, its entry in the symbol table will now contain that address.

During linking if the linker detects errors such as mis-spelling the name of a library function in the source code, or using the incorrect number or type of parameters for a function it stops the linking process and doesn't create the binary executable file.

Once the EXE file is produced by the linker it would be loaded by 'Loader' in memory and after calling some platform specific initialization functions, **main( )** would be called.

Figure 7.2 summarizes the role played by each processor program during the build process.

Processor	Input	Output
Editor	Program typed from keyboard	C source code containing program and preprocessor commands
Prepro-cessor	C source code file	Source code file with the preprocessing commands properly sorted out
Compiler	Source code file with preprocessing commands sorted out	Assemby language code
Assembler	Assembly language code	Relocatable Oject code in machine language
Linker	Object code of our program and object code of library functions	Executable code in machine language
Loader	Executable file	

Figure 7.2

One final word before we end this topic. The object file format created by compilers targeted for Windows and those targeted for Linux are different. Under Windows the object file's format is called Portable Executable (PE) file format. As against this, under Linux the popularly used object file formats are Common Object File Format (COFF) or Executable and Linking Format (ELF).

Summary

(a) The preprocessor directives enable the programmer to write programs that are easy to develop, read, modify and transport to a different computer system.

(b) We can make use of various preprocessor directives such as **#define, #include, #ifdef - #else - #endif, #if** and **#elif** in our program.

(c) The directives like **#undef** and **#pragma** are also useful although they are seldom used.

Exercise

[A] Answer the following:

(a) What is a preprocessor directive

1. a message from compiler to the programmer
2. a message from compiler to the linker
3. a message from programmer to the preprocessor
4. a message from programmer to the microprocessor

(b) Which of the following are correctly formed **#define** statements:

```
#define  INCH PER FEET  12
#define  SQR (X)  ( X * X )
#define  SQR(X)  X * X
#define  SQR(X)  ( X * X )
```

(c) State True or False:

1. A macro must always be written in capital letters.
2. A macro should always be accomodated in a single line.
3. After preprocessing when the program is sent for compilation the macros are removed from the expanded source code.
4. Macros with arguments are not allowed.
5. Nested macros are allowed.
6. In a macro call the control is passed to the macro.

(d) How many **#include** directives can be there in a given program file?