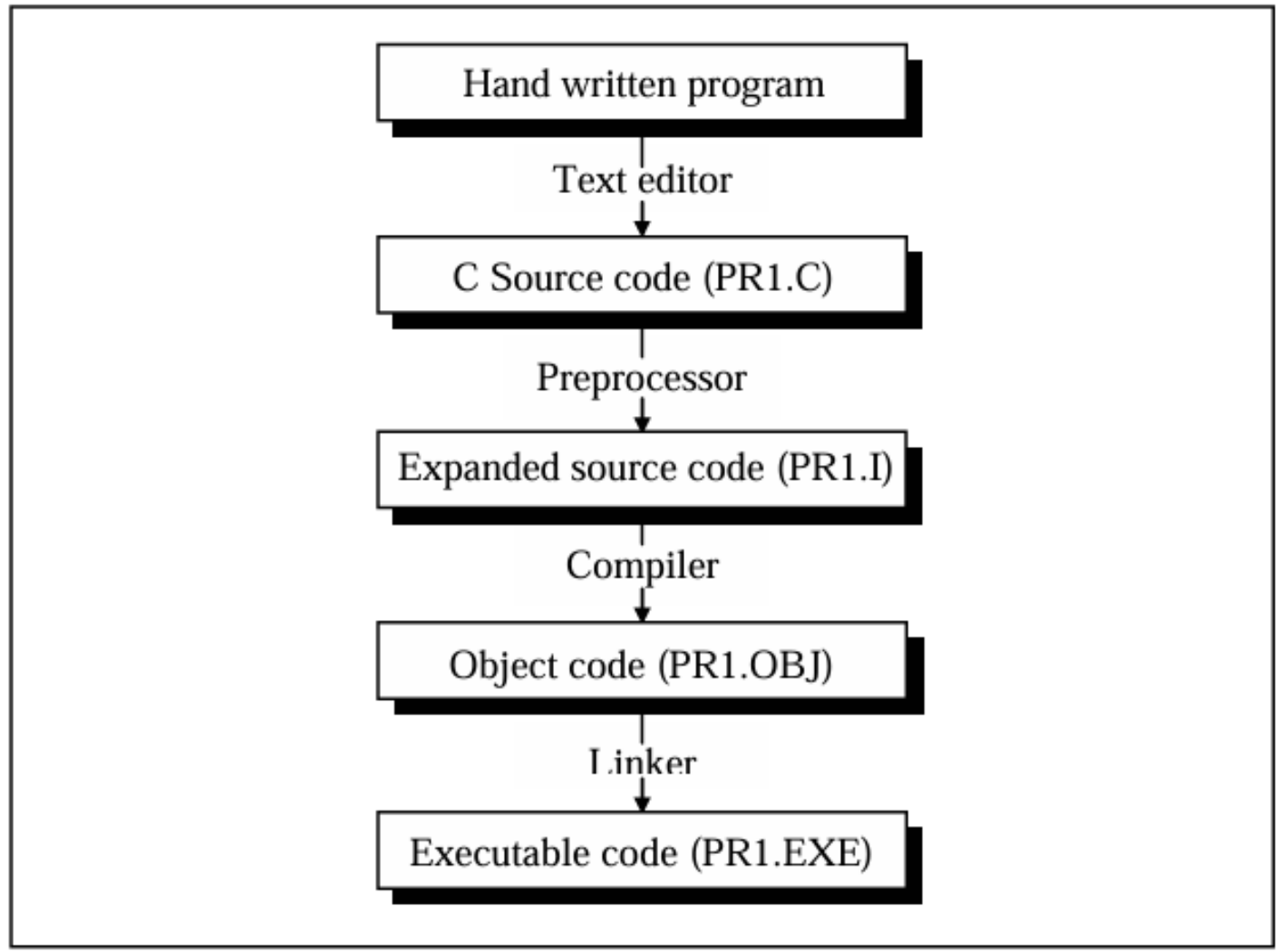
The C Preprocessor

- C preprocessor is exactly what its name implies.
- It is a program that processes our source program before it is passed to the compiler
- There are several steps involved from the stage of writing a C program to the stage of getting it executed.
- Figure 1 shows these different steps along with the files created during each stage.
- You can observe from the figure that our program passes through several processors before it is ready to be executed.
- The input and output to each of these processors is shown in Figure 2


Hand written program
Text editor
C Source code (PR1.C)
Preprocessor
Expanded source code (PR1.I)
Compiler
Object code (PR1.OBJ)
Linker
Executable code (PR1.EXE)

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	Relocatable object code
Linker	Relocatable object code and the standard C library functions	Executable code in machine language

Features of preprocessor

- 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 the first function definition.
- We would learn the following preprocessor directives here:
- (a) Macro expansion
- (b) File inclusion
- (c) Conditional Compilation
- (d) Miscellaneous directives

Macro Expansion

- #define UPPER 25
- This statement is called ‘macro definition’ or more commonly, just a ‘macro’
- During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25

```
#define UPPER 25
main( )
{
      int  i ;
      for ( i = 1 ; i <= UPPER ; i++ )
            printf ( "\n%d", i ) ;
}
```

- Here is another example of macro definition.
- 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.

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

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

Compilation process

- 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.

Points to note

- 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.

Reason for using macro

- Improve readability of the program-
- 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.

Reason for using macro

- If the value of the constant changes, it needs to be changed only once i.e. in the macro definition
- 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
- This convenience may not matter for small programs shown above, but with large programs macro definitions are almost indispensable.

Variable v/s Macro

- Same can be achieved by defining pi as a variable name
- However, we prefer macro in such cases for three reasons:
- 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.

#define for operators

- 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" ) ;
}
```

#define for condition

- 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" ) ;
}
```

#define for C statement

- A #define directive could be used to replace even an entire C statement

```
#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.

```
#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
```

Replacing of macro arguments

- 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 ) ;
}
```

Example

- 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" ) ;

}
```

Some important points

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

- 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.

- The macro was expanded into
- j = 64 / 4 * 4 ; which yielded 64.

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

Macros versus Functions

- In a macro call the preprocessor replaces the macro template with its macro expansion
- 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
- Usually macros make the program run faster but increase the program size,
- whereas functions make the program smaller and compact

Macros versus Functions

- 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

Conclusion

- 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 to use this feature

- It can be used in two cases:
- 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

- 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.

Header File

- 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.

Ways of writing #include statement

- 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.c" – This command would look for the file goto.c 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.c> - This command would look for the file goto.c 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.

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.

Usage

- 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.

Solution

- 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.

Usage 2

- 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
}
```

Example

- When you compile this program it would compile only the code suitable for a Motorola PC and the common code.
- This is because the macro INTEL has not been defined.
- If you want to run your program on an Intel PC, just add a statement at the top saying,
- #define INTEL

#ifndef

- 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
}
```

#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.
- If the expression, TEST <= 5 evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled.
- If we so desire we can have nested conditional compilation directives.

```
main( )
{
    #if TEST <= 5
        statement 1 ;
        statement 2 ;
        statement 3 ;
    #else
        statement 4 ;
        statement 5 ;
        statement 6 ;
    #endif
}
```