# C Preprocessors and Macros

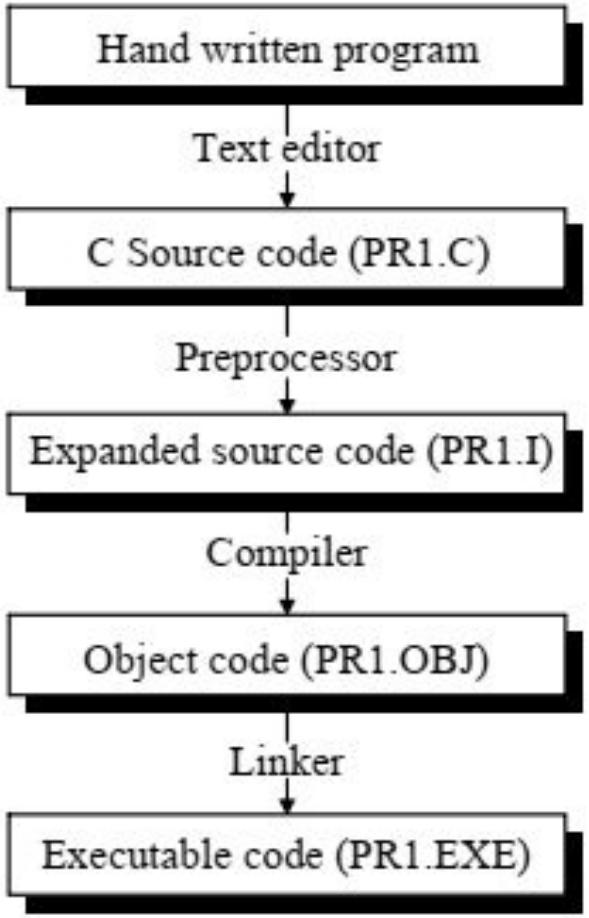
# PRE-PROCESSORS

- The C preprocessor 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.

# Features of C 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:
  - Macro expansion
  - File inclusion
  - Conditional Compilation
  - Miscellaneous directives

# Program Flow




| 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 |

# 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’.

# Macro Expansion

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’.

# Macro Expansion Procedure

- 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.

# General Conventions of Macro

- In C programming it is customary to use capital letters for macro template.
- 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.

# Examples of #define

```
#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 AND &&
#define ARANGE ( a > 25 AND a < 50 )
main( )
{
    int a = 30 ;
    if ( ARANGE )
    printf ( "within range" ) ;
    else
    printf ( "out of range" ) ;
}
```

# EXAMPLES OF #DEFINE

```
#define FOUND printf("The Yankee Doodle Virus");
main( )
{
   char signature ;
   if ( signature == 'Y' )
   FOUND
   else
   printf ( "Safe... as yet !" ) ;
}
```

# Macros with Arguments

```
#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

# Macros with Arguments

- Be careful not to leave a blank between the macro template and its argument while defining the macro.

```
#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 = 16

whereas, what we expected was j = 4.

# Macros with Arguments

- Macros can be split into multiple lines, with a '\' (back slash) present at the end of each line.

```
#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 ) ;
}
```

# 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.
- 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.

# File Inclusion

- 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.

# File Inclusion

- Actually there exist two ways to write #include statement. These are:
  - #include "filename"
  - #include <filename>

| | |
|---|---|
| #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. |

# Conditional Compilation

```
#ifdef macroname
    statement 1 ;
    statement 2 ;
    statement 3 ;
#endif

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

```
main( )
{
#ifdef OKAY
statement 1 ;
statement 2 ; /* detects virus */
statement 3 ;
statement 4 ; /* specific to stone
virus */
#endif
statement 5 ;
statement 6 ;
statement 7 ;
}
```

# #IF AND #ELIF DIRECTIVES

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

```
#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
```

# 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

- 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.
  - **#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).
  - **#pragma warn**: This directive tells the compiler whether or not we want to suppress a specific warning.