The Preprocessor directives

The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of preprocessor command line or directives.

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated
by the statements. After taking appropriate actions as directed by preprocessor, the source program is handed over to the compiler.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the new-line character at the end of the line by a backslash (\).
The preprocessor directives can be divided into following categories.

1. Macro substitution
2. File Inclusion
3. Compiler control Directives
4. Line control
5. Error Directives

1. Macro Substitution:

In macro substitution, an IDENTIFIER is replaced by pre-defined string. The preprocessor accomplishes this job under the direction of **#define** directives which is called macro. The syntax for macro definition is:

```
#define  IDENTIFIER  string
```

If this line is included at the beginning of the program, preprocessor replaces IDENTIFIER in source code by string defined in macro. The IDENTIFIER must be C/C++ valid name.

There are three different forms of macro substitutions:

a. Simple macro substitution
b. Argumented macro substitution
c. Nested macro substitution

- ### A simple macro:

A simple macro is generally used to define symbolic constants.
example:

```
#define PI  3.14159
#define TRUE  1
#define CITY    "Kathmandu"
```

```
        #define TWO_PI       2*(22/7)
        #define BLANK_LINE   cout<<"\n"
        #define  INCREMENT    ++
        #define  START       {
        #define  END         }
etc.
```

The Uppercase letter for IDENTIFIER is conventions for defining Constants.

- **Argumented macro :**

We can define complex macro with arguments. The syntax is

```
#define IDENTIFIER(a1,a2,…..,an)     string
```

✓ There is no space between IDENTIFIER and parenthesis .
✓ a1,a2,….,a_n are macro formal parameters.
✓ When a macro is called , the preprocessor substitutes the string, replacing the formal parameters with actual parameters(passed when calling macro)

Example:

```
#define   SQUARE(x)  ((x)*(x))
#define   CUBE(x)    ((x)*(x)*(x))
```

We can call these macro in our source program as

Area = PI*SQUARE(radius);

Which is equivalent to :

Area = PI*((radius)*(radius)); /*substitution of macro */

Other examples of arguemented macro:

```
#define   MAX(a,b)      (((a)>(b))? (a): (b))
#define   MIN(a,b)      (((a)<(b))? (a) : (b))
#define   MODVAL(x)      (((x)>0) ? (x) : (-(x)))
#define   RECIPROCAL(x)   (1.0/(x))
etc.
```

- **Nesting macro**:

We can use one pre-defined macro to define new macro. This is called nesting of macro definition.

Example:

```
#define   PI   3.14159
#define   TWO_PI    2*PI
#define   SQUARE(x)  ((x)*(x))
#define   CUBE(x)    ((x)*SQUARE(x))
#define   SIX_POWER(x)    ((CUBE(x))*(CUBE(x)))
```

✓ Given the definition of macro MAX(a,b) , We can use nested call of macro to fine maximum of three numbers a,b,c as:

```
MAX(a,MAX(a,b))
```

- To define multiple lines we can use \ character at the end of each line:

Undifining a Macro:

Defined macros are not affected by block structure. A macro lasts until it is undefined with the `#undef` preprocessor directive: A defined macro can be undefined , using the statement:

#undef IDENTIFIER

it is useful when to restrict the definition only to a particular part of the program.
e.g.

```
#define SIZE 100
int table1[SIZE];
cout<<SIZE;  // gives 100
#undef SIZE
#define SIZE 200
int table2[SIZE];
cout<<SIZE;  // gives 200
#undef SIZE
```

This would generate the same code as:

```
int table1[100];
cout<<100;
int table2[200];
cout<<200;

After final #undef command , if we place a line as
cout<<SIZE;  // error: undefined symbol SIZE
```

# and ## :

The argumented (function) macro definition accept two special operators (# and ##) in the replacement sequence. If the operator # is used before a parameter is used in the replacement sequence, that parameter is replaced by a string literal (as if it were enclosed between double quotes)

```
#define str(x) #x
cout<<str(test);
```

This would be translated into:

```
cout << "test";
```

The operator ## concatenates two arguments leaving no blank spaces between them:

```
#define glue(a,b) a ## b
glue(c,out) << "test";
```

This would also be translated into:

```
cout << "test";
```

2. File inclusion directives: (#include directives)

A file which contains the pre-defined macros and functions can be included in our source program using `#include` directive. If we include pre-written file in our source program we need not to rewrite the functions and macro which are defined in that file.
The syntax for file inclusion directive is :

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

➔ When `#include<filename>` is used, the file inside < > is searched in the standard directory and included in the source program where it is written. If file is not find an error is reported.
➔ When `#include "filename"` is used , the file inside " " is searched first in the current directory and then standard directory included in the source program where it is written. If file is not find an error is reported.

A simple example of file inclusion:

Consider a file named **test.cpp** which includes the definition of functions for square and cube and macro to find the maximum of two numbers , and constant PI.

```
/*  test.cpp */

#define  PI    3.14
#define  MAX(x,y)   ( ( (x) > (y) ) ? (x) : (y) )

int square(int x) /*square function */
{
  return x*x;
}

int cube(int x)     /*cube function */
{
  return (x*x*x);
}
```

Now we have to write a program that uses functions and macros defined in `test.cpp`. the source program is as:
/*source.cpp*/

```
#include<iostream.h>
#include "test.c"
void main()
{
   int x =10, y=20;
   cout<<"The square of"<<x<<" is "<<square(x))<<endl;
   cout<<"The cube of "<<x<<" is "<<cube(x))<<endl;
   cout<<"The larger no is "<<MAX(a,b))<<endl;
}
```

when we run the source file source.cpp the output will be

```
The square of 10 is 100
The cube of 10 is 1000
The larger no is 20.
```

3. Compiler control directives(Conditional Inclusion) : (#ifdef, #ifndef, #if, #endif, #else and #elif)

These directives allow to include or discard part of the code of a program if a certain condition is met. Conditional compilation enables the programmer to control the execution of preprocessor directive and the compilation of the program code. Each of the conditional preprocessor directive evaluates a constant integer expression.

The conditional preprocessor directive construct is similar to if-else construct.
`#ifdef` allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

In this case, the line of code `int table[TABLE_SIZE];` is only compiled if `TABLE_SIZE` was previously defined with `#define`, independently of its value. If it was not defined, that line will not be included in the program compilation.

`#ifndef` serves for the exact opposite: the code between `#ifndef` and `#endif` directives is only compiled if the specified identifier has not been previously defined. For example:

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

In this case, if when arriving at this piece of code, the `TABLE_SIZE` macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the `#define` directive would not be executed.

The `#if, #else and #elif (i.e., "else if")` directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if or #elif` can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
```

```
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

Notice how the whole structure of `#if, #elif and #else` chained directives ends with `#endif`.

The behavior of `#ifdef and #ifndef` can also be achieved by using the special operators `defined and !defined` respectively in any `#if or #elif` directive:

```
#if !defined TABLE_SIZE
#define TABLE_SIZE 100
#elif defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#endif
int table[TABLE_SIZE];
```

4. Line control (#line)

When we compile a program and some error happen during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

```
#line number "filename"
```

Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

`"filename"` is an optional parameter that allows to redefine the file name that will be shown. For example:

```
#line 20 "TEST.CPP"
int a?;
```

This code will generate an error that will be shown as:

```
Error TEST.CPP 20:Declaration syntax error in function main()
```

5. Error directive (#error)

This directive aborts the compilation process when it is found, generating a compilation the error that can be specified as its parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

This example aborts the compilation process if the macro name `__cplusplus` is not defined (this macro name is defined by default in all C++ compilers).

```
#ifndef SIZE
#error Array size is required!
#endif
```

In this code section, if `SIZE` is not defined, the compiler generates an error message as:

```
Error Directive:Array size is required! in function main()
```

Predefined macro names

The following macro names are defined at any time:

macro	value
__LINE__	Integer value representing the current line in the source code file being compiled.
__FILE__	A string literal containing the presumed name of the source file being compiled.
__DATE__	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
__cplusplus	An integer value. All C++ compilers have this constant defined to some value. If the compiler is fully compliant with the C++ standard its value is equal or greater than 199711L depending on the version of the standard they comply.

```
// standard macro names TEST.CPP
#include <iostream.h>
int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " << __FILE__ <<".”<<endl;
  cout << "Its compilation began " << __DATE__;
  cout << " at " << __TIME__ << ".”<<endl;
  cout << "The compiler gives a __cplusplus value of " << __cplusplus;
  return 0;
}
```

```
The output of the code is like this:

This is the line number 4 of file TEST.CPP.
Its compilation began Jun 10 2009 at 16:52:05.
The compiler gives a __cplusplus value of 1
```