

C++ PREPROCESSOR

http://www.tutorialspoint.com/cplusplus/cpp_preprocessor.htm

Copyright © tutorialspoint.com

The preprocessors are the directives, which give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).

You already have seen a **#include** directive in all the examples. This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. Let us see important directives:

The #define Preprocessor:

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is:

```
#define macro-name replacement-text
```

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example:

```
#include <iostream>
using namespace std;

#define PI 3.14159

int main ()
{
    cout << "Value of PI :" << PI << endl;

    return 0;
}
```

Now, let us do the preprocessing of this code to see the result, assume we have source code file, so let us compile it with -E option and redirect the result to test.p. Now, if you will check test.p, it will have lots of information and at the bottom, you will find the value replaced as follows:

```
$gcc -E test.cpp > test.p

...
int main ()
{
    cout << "Value of PI :" << 3.14159 << endl;

    return 0;
}
```

Function-Like Macros:

You can use #define to define a macro which will take argument as follows:

```
#include <iostream>
using namespace std;

#define MIN(a,b) ((a)<(b)) ? a : b

int main ()
{
```

```

int i, j;
i = 100;
j = 30;
cout <<"The minimum is " << MIN(i, j) << endl;

return 0;
}

```

If we compile and run above code, this would produce the following result:

```
The minimum is 30
```

Conditional Compilation:

There are several directives, which can use to compile selectively portions of your program's source code. This process is called conditional compilation.

The conditional preprocessor construct is much like the if selection structure. Consider the following preprocessor code:

```

#ifdef NULL
#define NULL 0
#endif

```

You can compile a program for debugging purpose and can debugging turn on or off using a single macro as follows:

```

#ifdef DEBUG
cerr <<"Variable x = " << x << endl;
#endif

```

causes the **cerr** statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive `#ifdef DEBUG`. You can use `#if 0` statement to comment out a portion of the program as follows:

```

#if 0
code prevented from compiling
#endif

```

Let us try the following example:

```

#include <iostream>
using namespace std;
#define DEBUG

#define MIN(a,b) ((a)<(b)) ? a : b)

int main ()
{
    int i, j;
    i = 100;
    j = 30;
#ifdef DEBUG
    cerr <<"Trace: Inside main function" << endl;
#endif

    #if 0
        /* This is commented part */
        cout << MKSTR(HELLO C++) << endl;
    #endif

    cout <<"The minimum is " << MIN(i, j) << endl;

#ifdef DEBUG
    cerr <<"Trace: Coming out of main function" << endl;
#endif
    return 0;
}

```

If we compile and run above code, this would produce the following result:

```
Trace: Inside main function
The minimum is 30
Trace: Coming out of main function
```

The # and ## Operators:

The # and ## preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes.

Consider the following macro definition:

```
#include <iostream>
using namespace std;

#define MKSTR( x ) #x

int main ()
{
    cout << MKSTR(HELLO C++) << endl;

    return 0;
}
```

If we compile and run above code, this would produce the following result:

```
HELLO C++
```

Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line:

```
cout << MKSTR(HELLO C++) << endl;
```

into the following line:

```
cout << "HELLO C++" << endl;
```

The ## operator is used to concatenate two tokens. Here is an example:

```
#define CONCAT( x, y ) x ## y
```

When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows.

```
#include <iostream>
using namespace std;

#define concat(a, b) a ## b
int main()
{
    int xy = 100;

    cout << concat(x, y);
    return 0;
}
```

If we compile and run above code, this would produce the following result:

```
100
```

Let us see how it worked. It is simple to understand that the C++ preprocessor transforms:

```
cout << concat(x, y);
```

into the following line:

```
cout << xy;
```

Predefined C++ Macros:

C++ provides a number of predefined macros mentioned below:

Macro	Description
__LINE__	This contain the current line number of the program when it is being compiled.
__FILE__	This contain the current file name of the program when it is being compiled.
__DATE__	This contains a string of the form month/day/year that is the date of the translation of the source file into object code.
__TIME__	This contains a string of the form hour:minute:second that is the time at which the program was compiled.

Let us see an example for all the above macros:

```
#include <iostream>
using namespace std;

int main ()
{
    cout << "Value of __LINE__ : " << __LINE__ << endl;
    cout << "Value of __FILE__ : " << __FILE__ << endl;
    cout << "Value of __DATE__ : " << __DATE__ << endl;
    cout << "Value of __TIME__ : " << __TIME__ << endl;

    return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Value of __LINE__ : 6
Value of __FILE__ : test.cpp
Value of __DATE__ : Feb 28 2011
Value of __TIME__ : 18:52:48
```