



## Section 5.5

# Preprocessor directives

Preprocessor directives are orders that we include within the code of our programs that are not instructions for the program itself but for the preprocessor. The preprocessor is executed automatically by the compiler when we compile a program in C++ and is the one in charge to make the first verifications and digestions of the program's code.

All these directives must be specified in a single line of code and they do not have to include an ending semicolon ;.

## #define

At the beginning of this tutorial we already spoken about a preprocessor directive: `#define`, that serves to generate what we called *defined constants* or *macros* and whose form is the following:

```
#define name value
```

Its function is to define a macro called *name* that whenever it is found in some point of the code is replaced by *value*. For example:

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
char str2[MAX_WIDTH];
```

It defines two strings to store up to 100 characters.

`#define` can also be used to generate macro functions:

```
#define getmax(a,b) a>b?a:b
int x=5, y;
y = getmax(x,2);
```

after the execution of this code `y` would contain 5.

## #undef

`#undef` fulfills the inverse functionality than `#define`. What it does is to eliminate from the list of defined constants the one that has the name passed as parameter to `#undef`:

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
#undef MAX_WIDTH
#define MAX_WIDTH 200
char str2[MAX_WIDTH];
```

## #ifdef, #ifndef, #if, #endif, #else and #elif

These directives allow to discard part of the code of a program if a certain condition is not fulfilled.

`#ifdef` allows that a section of a program is compiled only if the *defined constant* that is specified as parameter has been defined, independently of its value. Its operation is:

```
#ifdef name
// code here
#endif
```

For example:

```
#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif
```

In this case, the line `char str[MAX_WIDTH];` is only considered by the compiler if the *defined constant* `MAX_WIDTH` has been previously defined, independently of its value. If it has not been defined, that line will not be included in the program.

`#ifndef` serves for the opposite: the code between the `#ifndef` directive and the `#endif` directive is only compiled if the constant name that is specified has not been defined previously. For example:

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 100
#endif
char str[MAX_WIDTH];
```

In this case, if when arriving to this piece of code the *defined constant* `MAX_WIDTH` has not yet been defined it would be defined with a value of 100. If it already existed it would maintain the value that it had (because the `#define` statement won't be executed).

The `#if`, `#else` and `#elif` (*elif = else if*) directives serve for that the portion of code that follows is compiled only if the specified condition is met. The condition can only serve to evaluate constant expressions. For example:

```
#if MAX_WIDTH>200
#undef MAX_WIDTH
#define MAX_WIDTH 200

#elif MAX_WIDTH<50
#undef MAX_WIDTH
#define MAX_WIDTH 50

#else
#undef MAX_WIDTH
#define MAX_WIDTH 100
#endif

char str[MAX_WIDTH];
```

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

## #line

When we compile a program and there happens any errors during the compiling process, the compiler shows the error that have happened preceded by the name of the file and the line within the file where it has taken place.

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 it appears when an error takes place. Its form is the following one:

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

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

*filename* is an optional parameter that serves to replace the file name that will be shown in case of error from this directive until other one changes it again or the end of the file is reached. For example:

```
#line 1 "assigning variable"  
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 1.

## #error

This directive aborts the compilation process when it is found returning the error that is specified as parameter:

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

This example aborts the compilation process if the *defined constant* `__cplusplus` is not defined.

## #include

This directive has also been used assiduously in other sections of this tutorial. When the preprocessor finds an `#include` directive it replaces it by the whole content of the specified file. There are two ways to specify a file to be included:

```
#include "file"  
#include <file>
```

The only difference between both expressions is in the directories in which the compiler is going to look for the file. In the first case in that the file is specified between quotes the file is looked for from the same directory in which the file that includes the directive is, and only in case that it is not there the compiler looks for in the default directories where it is configured to look for the standard header files.

In case that the file name is included enclosed between angle-brackets `<>` the file is directly looked for in the default directories where the compiler is configured to look for the standard header files.

## #pragma

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`.

© The C++ Resources Network, 2000-2001 - All rights reserved

[Previous:](#)    [Next:](#)  
[5-4. Advances classes type casting.](#) [index](#) [6-1. Input/Output with files.](#)