Preprocessor directives

Preprocessor directives are lines included in the code of programs preceded by a hash sign (`#`). These lines are not program statements but directives for the *preprocessor*. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

预处理器指令是包含在程序代码中前面有#号(#)的行。这些行不是程序语句，而是预处理器的指令。预处理器在实际编译代码之前检查代码，并在常规语句实际生成代码之前解析所有这些指令。

These *preprocessor directives* extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. 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 newline character at the end of the line by a backslash (`\`).

这些预处理器指令只能扩展到一行代码。一旦发现换行符，预处理器指令就是ends。在预处理器指令的末尾不能使用分号(;)。预处理器指令可以扩展到不止一行的唯一方法是在换行符行尾前面加一个反斜杠()。

macro definitions (#define, #undef)

To define preprocessor macros we can use `#define`. Its syntax is:

要定义预处理器宏，可以使用#define。它的语法是:

`#define identifier replacement`

When the preprocessor encounters this directive, it replaces any occurrence of `identifier` in the rest of the code by `replacement`. This `replacement` can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of `identifier` by `replacement`.

当预处理器遇到这个指令时，它将替换代码其余部分中出现的任何标识符。这个替换可以是表达式、语句、块或任何东西。预处理器不理解c++本身，它只是用replacement替换任何出现的标识符。

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

After the preprocessor has replaced `TABLE_SIZE`, the code becomes equivalent to:

```
int table1[100];
int table2[100];
```

`#define` can work also with parameters to define function macros:

#define也可以与参数一起定义函数宏:

```
#define getmax(a,b) a>b?a:b
```

This would replace any occurrence of `getmax` followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as you would expect if it was a function:

这将用替换表达式替换getmax后面跟着两个参数的任何情况，但也会用它的标识符替换每个参数，就像你期望的那样，如果它是一个函数:

```
// function macro
#include <iostream>
using namespace std;

#define getmax(a,b) ((a)>(b)?(a):(b))

int main()
{
  int x=5, y;
  y= getmax(x,2);
  cout << y << endl;
  cout << getmax(7,x) << endl;
  return 0;
}
```

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

已定义的宏不受块结构的影响。一个宏会一直存在，直到它被#undef预处理器指令定义:

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

This would generate the same code as:

这将生成如下代码:

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

Function macro definitions accept two special operators (`#` and `##`) in the replacement sequence:

函数宏定义在替换序列中接受两个特殊操作符(#和##):

The operator `#`, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):

后面跟着参数名的操作符#被包含传入参数(就像用双引号括起来的)的字符串字面量所取代:

```
#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:

操作符' ## '连接两个参数，它们之间不留下空格:

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

This would also be translated into:

这句话也可以译成:

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

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature. But, be careful: code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++.

因为预处理器替换发生在任何c++语法检查之前，宏定义可能是一个棘手的特性。但是，要注意:严重依赖复杂宏的代码可读性会变差，因为在很多情况下，预期的语法与程序员在c++中预期的正常表达式不同。

Conditional inclusions (#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.

这些指令允许在满足特定条件时包含或丢弃程序的部分代码。

`#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只允许在定义了作为参数的宏的情况下编译程序的一部分，而不管它的值是多少。例如:

```
#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.

在这种情况下，代码行int table[TABLE_SIZE];仅当TABLE_SIZE之前用#define定义时才编译，与它的值无关。如果没有定义，这一行将不会包含在程序编译中。

`#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的作用正好相反:#ifndef和#endif指令之间的代码只有在之前没有定义指定标识符的情况下才会被编译。例如:

```
#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.

在这种情况下，如果当到达这段代码时，TABLE_SIZE宏还没有定义，那么它的值将被定义为100。如果它已经存在，它将保留之前的值，因为#define指令将不会被执行。

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， #else和#elif(即"else if")指令用于指定要满足的一些条件，以便它们包围的部分代码被编译。#if或#elif后面的条件只能计算常量表达式，包括宏表达式。例如:

```
#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 entire structure of `#if`, `#elif` and `#else` chained directives ends with `#endif`.

注意，#if、#elif和#else链式指令的整个结构是如何以#endif结尾的。

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:

#ifdef和#ifndef的行为也可以通过使用在任意#if或#elif指令中定义的特殊操作符来实现:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

Line control (#line)

When we compile a program and some error happens 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.

当我们编译一个程序时，在编译过程中发生了一些错误，编译器会显示一个错误消息，其中引用了发生错误的文件的名称和行号，因此更容易找到生成错误的代码。

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指令允许我们控制这两件事，代码文件中的行号，以及当错误发生时我们希望出现的文件名。它的格式是:

`#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.

其中number是将分配给下一个代码行的新行号。从这一点开始，连续行的行数将一个接一个地增加。

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

"filename"是一个可选参数，允许重新定义将显示的文件名。例如:

```
#line 20 "assigning variable"
int a?;
```

This code will generate an error that will be shown as error in file `"assigning variable"`, line 20.

这段代码将生成一个错误，在文件"assign variable"的第20行显示为错误。

Error directive (#error)

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

当找到它时，这个指令会终止编译过程，生成一个编译错误，可以指定为它的参数:

```
#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).

如果没有定义宏名__cplusplus(这个宏名在所有c++编译器中都是默认定义的)，这个例子会终止编译过程。

Source file inclusion (#include)

This directive has been used assiduously in other sections of this tutorial. When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified header or file. There are two ways to use `#include`:

这个指令在本教程的其他部分已经被认真地使用了。当预处理器找到一个#include指令时，它会用指定头文件或文件的全部内容替换它。#include有两种使用方法:

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

In the first case, a *header* is specified between angle-brackets `<>`. This is used to include headers provided by the implementation, such as the headers that compose the standard library (`iostream`, `string`,...). Whether the headers are actually files or exist in some other form is *implementation-defined*, but in any case they shall be properly included with this directive.

在第一种情况下，头文件在尖括号<>之间指定。这用于包含实现提供的头文件，比如组成标准库的头文件(iostream, string，...)。头文件是否真的是文件或以其他形式存在是由实现定义的，但在任何情况下，它们都应该被正确地包含在这个指令中。

The syntax used in the second `#include` uses quotes, and includes a *file*. The *file* is searched for in an *implementation-defined* manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a *header* inclusion, just as if the quotes (`""`) were replaced by angle-brackets (`<>`).

第二个#include中使用的语法使用引号，并包含一个文件。以实现定义的方式搜索文件，通常包括当前路径。在没有找到文件的情况下，编译器将该指令解释为头文件包含，就像引号("")被尖括号(<>)替换一样。

Pragma directive (#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`.

该指令用于向编译器指定不同的选项。这些选项是特定于您使用的平台和编译器的。有关可以使用#pragma定义的可能参数的更多信息，请参阅编译器的手册或参考资料。

If the compiler does not support a specific argument for `#pragma`, it is ignored - no syntax error is generated.

如果编译器不支持#pragma的特定参数，它将被忽略——不会生成语法错误。

Predefined macro names

The following macro names are always defined (they all begin and end with two underscore characters, `_`):

下面的宏名称总是被定义(它们都以两个下划线' _ '开始和结束):

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. Its value depends on the version of the standard supported by the compiler: **`199711L`**: ISO C++ 1998/2003**`201103L`**: ISO C++ 2011Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above.
`__STDC_HOSTED__`	`1` if the implementation is a *hosted implementation* (with all standard headers available) `0` otherwise.

The following macros are optionally defined, generally depending on whether a feature is available:

下面的宏是可选定义的，通常取决于一个特性是否可用:

macro	value
`__STDC__`	In C: if defined to `1`, the implementation conforms to the C standard. In C++: Implementation defined.
`__STDC_VERSION__`	In C: **`199401L`**: ISO C 1990, Ammendment 1 **`199901L`**: ISO C 1999 **`201112L`**: ISO C 2011In C++: Implementation defined.
`__STDC_MB_MIGHT_NEQ_WC__`	`1` if multibyte encoding might give a character a different value in character literals
`__STDC_ISO_10646__`	A value in the form `yyyymmL`, specifying the date of the Unicode standard followed by the encoding of `wchar_t` characters
`__STDCPP_STRICT_POINTER_SAFETY__`	`1` if the implementation has *strict pointer safety* (see `get_pointer_safety`)
`__STDCPP_THREADS__`	`1` if the program can have more than one thread

Particular implementations may define additional constants.

特定的实现可以定义额外的常量。

For example:

```
// standard macro names
#include <iostream>
using namespace std;

int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " << __FILE__ << ".\n";
  cout << "Its compilation began " << __DATE__;
  cout << " at " << __TIME__ << ".\n";
  cout << "The compiler gives a __cplusplus value of " << __cplusplus;
  return 0;
}
```