Advanced Concepts:

23. Preprocessor 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 pound 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.

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 newline character at the end of the line by a backslash (\).

macro definitions (#define, #undef)

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

```
#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++, it simply replaces any occurrence of `identifier` by `replacement`.

```
#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];
```

This use of #define as constant definer is already known by us from previuos tutorials, but `#define` can work also with parameters to define function macros:

```
#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:

```
// 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;
}
```

```
5
7
```

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

```
#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:

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

Function macro definitions 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";
```

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 may result obscure to other programmers, since the syntax they expect is on many occasions different from the regular expressions programmers expect in C++.

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.

`#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
int table[TABLE_SIZE];
```

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 "assigning variable"
int a?;
```

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

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).

Source file inclusion (#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 entire 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 the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.
If the file name is enclosed between angle-brackets <> the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

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`.

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

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.

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;
}
```

```
This is the line number 7 of file
/home/jay/stdmacronames.cpp.
Its compilation began Nov  1 2005
at 10:12:29.
The compiler gives a __cplusplus
value of 1
```