Modern C++ - The preprocessor

4. Preprocessor

Preprocessor directives

Preprocessing is the first steps of the C++ translation process.

1. Trigraph replacement (trigraph characters are replaced).
2. Line splicing (lines ending with escaped newlines are spliced).
3. Tokenization (and replacing comments with whitespace).
4. Macro expansion and directive handling.

Digraphs and trigraphs

The required character set of the programming language and the usual input devices (e.g. keyboard) may differ.

Not only in C++, C and Pascal also supports digraphs:

```
(.    -->   [
.)    -->   ]
(*    -->   {
*)    -->   }
```

C++ language trigraphs:

```
??=   -->   #
??/   -->   \
??'   -->   ^
??(   -->   [
??)   -->   ]
??!   -->   |
??<   -->   {
??>   -->   }
??-   -->   ~
```

```
printf("How are you??-I am fine")  -->  printf("How are you~I am fine")
printf("How are you???-I am fine")  -->  printf("How are you?~I am fine")

```

```
// Will the next line be executed??????/
a++;                  -->  // Will the next line be executed???? a++;

/??/
 * A comment *??/
/                      --> /* A comment */
```

Digraphs handled in tokenization (step 4.)

```
<:   -->   [
:>   -->   ]
<%   -->   {
%>   -->   }
%:   -->   #
```

Include directive handling

```
#include <filename>
```

Search filename in the standard compiler include path

```
#include "filename"
```

Search filename in the current source directory

```
$ g++ -I/usr/local/include/add/path1 -I/usr/local/include/add/path2 ...
```

Extend the include path from command line

The preprocessor replaces the line with the text of the *filename*.

```
#include <iostream>
int main(void)
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Macro definition

There are *object-like* and *function-like* macros.

Object-like macro:

```
#define <identifier>  <token-list>
```

Examples:

```
#define BUFSIZE    1024
#define PI         3.14159
#define USAGE_MSG  "Usage: command -flags args..."
#define LONG_MACRO struct MyType \
                   {             \
                     int data;   \
                   };
```

Usage:

```
char buffer[BUFSIZE];
fgets(buffer, BUFSIZE, stdin);
```

Function-like macro

```
#define <identifier>(<param-list>)  <token-list>
```

Examples:

```
#define FAHR2CELS(x)  ((5./9.)*(x-32))
#define MAX(a,b)  ((a) > (b) ? (a) : (b))
```

Usage:

```
c = FAHR2CELS(f);
x = MAX(x,-x);
x = MAX(++y,z);
```

Undefine macros

```
#undef BUFSIZE
```

Conditional compilation

In some cases (e.g. configuration parameters, platforms, etc.) we need to separate different compilation cases. We can do this with *conditional compilation*.

In the condition expression, all non-zero values mean **true**, zero value means **false**.

Sample: Configuration on debug level:

```
#if DEBUG_LEVEL > 2
  fprint("program was here %s %d\n", __FILE__, __LINE__);
#endif
```

Sample: configuration on OS platform:

```
#ifdef __unix__ /* __unix__ is usually defined by compilers for Unix */
#  include <unistd.h>
#elif defined _WIN32 /* _Win32 is usually defined for 32/64 bit Windows */
#  include <windows.h>
#endif
```

We can use complex (but preprocessor-time computable) expressions for the condition.

```
#if !(defined( __unix__ ) || defined (_WIN32) )
  // ...
#else
  // ...
#endif
```

Error

In some cases we want to force stop the compilation.

```
#if RUBY_VERSION == 190
# error 1.9.0 not supported
#endif
```

Line

Line directive is used mostly for debug purposes, it sets the line number for the compiler (and for the **LINE** macro).

```
#line 567
```

Header guards

Heared guard is the most frequently used preprocessor pattern in C++. Its purpose is to avoid multiple inclusion:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
/*
 * content of header file
 *
 */
#endif /* MY_HEADER_H */
```

Since C++11 we have an alternative solution:

```
#pragma ones
/*
 * content of header file
 *
 */
```

Standard defined macros

```
__FILE__
__LINE__
__DATE__
__TIME__

__STDC__
__STDC_VERSION__
__cplusplus
```

To check whether we have a standard C++ compiler which runs the preprocessor. Mostly used for library headers which can be included into either C or C++ code.

```
#ifdef __cplusplus
extern C {
#endif
// ...
#ifdef __cplusplus
}
#endif
```

Pragma

Pragmas have compiler-defined effect.

```
#pragma warning(disable:4786)
```

Token stringification

For very tricky macro-s we need to create a string from an originally non-string value (like a token or a substritution result of an earlier macro).

```
#define str(s) #s
#define BUFSIZE 1024
// ...
str(\n)        -->    "\n"
str(BUFSIZE)   -->    1024
```

Token concatenation

When two string literal ar separated with the **##** characters, the separator whitespaces are discarded and the string literals are concatenated.

This is used both for creating long string literals and for hacking template-like macros.

```
struct my_int_20_array
{
  int v[20];
};
struct my_int_30_array
{
  int v[30];
};
struct my_double_40_array
{
  double v[40];
};

#define DECLARE_ARRAY(NAME, TYPE, SIZE) \
typedef struct TYPE##_##SIZE##_array    \
{                                       \
  TYPE v[SIZE];                              \
                                             \
} NAME##_t;

DECLARE_ARRAY(yours,float,10);
yours_t x, y;
```