**Preprocessors**

Prior to compilation, the code file goes through a phase known as translation. A code file with translations applied to it is called a **translation unit**.

The most noteworthy of the translation phases involves the preprocessor. The preprocessor is best thought of as a separate program that manipulates the text in each code file.

When the preprocessor runs, it scans through the code file (from top to bottom), looking for preprocessor directives.

# Preprocessor directives

Preprocessor directives (often just called directives) are instructions that start with a # symbol and end with a newline (NOT a semicolon). These directives tell the preprocessor to perform specific particular text manipulation tasks. Note that the preprocessor does not understand C++ syntax -- instead, the directives have their own syntax (which in some cases resembles C++ syntax, and in other cases, not so much).

# Types of preprocessor directives

There are 4 main types of preprocessor directives:

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

# Macros

The #define directive can be used to create a macro. In C++, a macro is a rule that defines how input text is converted into replacement output text.

There are two basic types of macros:

* object-like macros
* function-like macros

## Object-like macros

can be defined in one of two ways:

#define identifier

#define identifier substitution\_text

The top definition has no substitution text, whereas the bottom one does. Because these are preprocessor directives (not statements), note that neither form ends with a semicolon.

Object-like macros don’t affect other preprocessor directives

Macros only cause text substitution for normal code. Other preprocessor commands are ignored. Consequently, the PRINT\_JOE in #ifdef PRINT\_JOE is left alone.

#define FOO 9 // Here's a macro substitution

#ifdef FOO // This FOO does not get replaced because it’s part of another preprocessor directive

std::cout << FOO; // This FOO gets replaced with 9 because it's part of the normal code

#endif

## function-like macros

Macros defined with arguments works similarly as functions.

Their use is generally considered dangerous, and almost anything they can do can be done by a normal function.

#include <iostream>

#define LIMIT 5 // object-like macro

#define RECT\_AREA(l, b) (l \* b) // function-like macro

int main() {

for (int i = 0; i < LIMIT; i++) {

std::cout << i << ' ';

}

std::cout << '\n';

int l1 = 10, l2 = 5, area = 0;

area = RECT\_AREA(l1, l2);

std::cout << "Area of rectangle is: " << area;

return 0;

}

Output:

0 1 2 3 4

Area of rectangle is: 50

# File Inclusion

When you #include a file, the preprocessor replaces the #include directive with the contents of the included file. The included contents are then preprocessed (along with the rest of the file), and then compiled.

#include <file\_name> // to include standard file

#include "filename" // to include user defined files

# Conditional Compilation

The conditional compilation preprocessor directives allow you to specify under what conditions something will or won’t compile. There are quite a few different conditional compilation directives.

The three that are used by far the most here: #ifdef, #ifndef, and #endif.

**#ifdef**

The #ifdef preprocessor directive allows the preprocessor to check whether an identifier has been previously #defined. If so, the code between the #ifdef and matching #endif is compiled. If not, the code is ignored.

**#ifndef**

#ifndef is the opposite of #ifdef, in that it allows you to check whether an identifier has NOT been #defined yet.

NOTE:

In place of #ifdef PRINT\_BOB and #ifndef PRINT\_BOB, you’ll also see **#if defined(PRINT\_BOB) and #if !defined(PRINT\_BOB**). These do the same, but use a **slightly more C++-style syntax**.

**#if 0**

* One more common use of conditional compilation involves using #if 0 to exclude a block of code from being compiled (as if it were inside a comment block).
* This also provides a convenient way to “comment out” code that contains multi-line comments (which can’t be commented out using another multi-line comment due to multi-line comments being non-nestable).

# Other Directives

## #undef Directive

The #undef directive is used to undefine an existing macro. This directive works as:

#undef LIMIT

Using this statement will undefine the existing macro LIMIT. After this statement every “#ifdef LIMIT” statement will evaluate to false.

## #pragma Directive

Pragma directive controls implementation-specific behavior of the compiler, such as disabling compiler warnings or changing alignment requirements. Any pragma that is not recognized is ignored.

**Non-standard pragmas**

The ISO C++ language standard does not require the compilers to support any pragmas. However, several non-standard pragmas are supported by multiple implementations.

### #pragma startup and #pragma exit

These directives helps us to specify the functions that are needed to run before program startup( before the control passes to main()) and just before program exit (just before the control returns from main()).

GCC does not supports #pragma startup or exit.

### #pragma once

If it appears in a header file, it indicates that it is only to be parsed once, even if it is (directly or indirectly) included multiple times in the same source file.

#ifndef LIBRARY\_FILENAME\_H

#define LIBRARY\_FILENAME\_H

// contents of the header

#endif /\* LIBRARY\_FILENAME\_H \*/

So that all but the first inclusion of the header in any translation unit are excluded from compilation. All modern compilers record the fact that a header file uses an include guard and do not re-parse the file if it is encountered again, as long as the guard is still defined.

With #pragma once, the same header appears as

#pragma once

// contents of the header

Unlike header guards, this pragma makes it impossible to erroneously use the same macro name in more than one file. On the other hand, since with #pragma once files are excluded based on their filesystem-level identity, this can't protect against including a header twice if it exists in more than one location in a project.

# The scope of defines

Directives are resolved before compilation, from top to bottom on a file-by-file basis.

Consider the following program:

#include <iostream>

void foo() {

#define MY\_NAME "Alex"

}

int main() {

std::cout << "My name is: " << MY\_NAME;

return 0;

}

Even though it looks like #define MY\_NAME “Alex” is defined inside function foo, the preprocessor won’t notice, as it doesn’t understand C++ concepts like functions. Therefore, this program behaves identically to one where #define MY\_NAME “Alex” was defined either before or immediately after function foo. For general readability, you’ll generally want to #define identifiers outside of functions.

Once the preprocessor has finished, all defined identifiers from that file are discarded. This means that directives are only valid from the point of definition to the end of the file in which they are defined. Directives defined in one code file do not have impact on other code files in the same project.

Consider the following example:

function.cpp:

#include <iostream>

void doSomething() {

#ifdef PRINT

std::cout << "Printing!";

#endif

#ifndef PRINT

std::cout << "Not printing!";

#endif

}

main.cpp:

void doSomething(); // forward declaration for function doSomething()

#define PRINT

int main() {

doSomething();

return 0;

}

Output:

Not printing!

Even though PRINT was defined in main.cpp, that doesn’t have any impact on any of the code in function.cpp (PRINT is only #defined from the point of definition to the end of main.cpp). This will be of consequence when we discuss header guards in a future lesson.

# References

<https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/>

<https://www.geeksforgeeks.org/cc-preprocessors/>

<https://en.cppreference.com/w/cpp/preprocessor>

# END