

Cpplib Internals

For GCC version 16.0.0 (pre-release)

(GCC)

Neil Booth

Copyright © 2000-2025 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

Conventions	1
The Lexer	3
Overview	3
Lexing a token	3
Lexing a line	5
Hash Nodes	9
Macro Expansion Algorithm	11
Internal representation of macros	11
Macro expansion overview	11
Scanning the replacement list for macros to expand	12
Looking for a function-like macro's opening parenthesis	13
Marking tokens ineligible for future expansion	13
Token Spacing	15
Line numbering	17
Just which line number anyway?	17
Representation of line numbers	17
The Multiple-Include Optimization	19
File Handling	21
Concept Index	23

Conventions

cpplib has two interfaces—one is exposed internally only, and the other is for both internal and external use.

The convention is that functions and types that are exposed to multiple files internally are prefixed with ‘_cpp_’, and are to be found in the file `internal.h`. Functions and types exposed to external clients are in `cpplib.h`, and prefixed with ‘cpp_’. For historical reasons this is no longer quite true, but we should strive to stick to it.

We are striving to reduce the information exposed in `cpplib.h` to the bare minimum necessary, and then to keep it there. This makes clear exactly what external clients are entitled to assume, and allows us to change internals in the future without worrying whether library clients are perhaps relying on some kind of undocumented implementation-specific behavior.

The Lexer

Overview

The lexer is contained in the file `lex.cc`. It is a hand-coded lexer, and not implemented as a state machine. It can understand C, C++ and Objective-C source code, and has been extended to allow reasonably successful preprocessing of assembly language. The lexer does not make an initial pass to strip out trigraphs and escaped newlines, but handles them as they are encountered in a single pass of the input file. It returns preprocessing tokens individually, not a line at a time.

It is mostly transparent to users of the library, since the library's interface for obtaining the next token, `cpp_get_token`, takes care of lexing new tokens, handling directives, and expanding macros as necessary. However, the lexer does expose some functionality so that clients of the library can easily spell a given token, such as `cpp_spell_token` and `cpp_token_len`. These functions are useful when generating diagnostics, and for emitting the preprocessed output.

Lexing a token

Lexing of an individual token is handled by `_cpp_lex_direct` and its subroutines. In its current form the code is quite complicated, with read ahead characters and such-like, since it strives to not step back in the character stream in preparation for handling non-ASCII file encodings. The current plan is to convert any such files to UTF-8 before processing them. This complexity is therefore unnecessary and will be removed, so I'll not discuss it further here.

The job of `_cpp_lex_direct` is simply to lex a token. It is not responsible for issues like directive handling, returning lookahead tokens directly, multiple-include optimization, or conditional block skipping. It necessarily has a minor rôle to play in memory management of lexed lines. I discuss these issues in a separate section (see [Lexing a line], page 5).

The lexer places the token it lexes into storage pointed to by the variable `cur_token`, and then increments it. This variable is important for correct diagnostic positioning. Unless a specific line and column are passed to the diagnostic routines, they will examine the `line` and `col` values of the token just before the location that `cur_token` points to, and use that location to report the diagnostic.

The lexer does not consider whitespace to be a token in its own right. If whitespace (other than a new line) precedes a token, it sets the `PREV_WHITE` bit in the token's flags. Each token has its `line` and `col` variables set to the line and column of the first character of the token. This line number is the line number in the translation unit, and can be converted to a source (file, line) pair using the line map code.

The first token on a logical, i.e. unescaped, line has the flag `BOL` set for beginning-of-line. This flag is intended for internal use, both to distinguish a '#' that begins a directive from one that doesn't, and to generate a call-back to clients that want to be notified about the start of every non-directive line with tokens on it. Clients cannot reliably determine this for themselves: the first token might be a macro, and the tokens of a macro expansion do not have the `BOL` flag set. The macro expansion may even be empty, and the next token on the line certainly won't have the `BOL` flag set.

New lines are treated specially; exactly how the lexer handles them is context-dependent. The C standard mandates that directives are terminated by the first unescaped newline character, even if it appears in the middle of a macro expansion. Therefore, if the state variable `in_directive` is set, the lexer returns a `CPP_EOF` token, which is normally used to indicate end-of-file, to indicate end-of-directive. In a directive a `CPP_EOF` token never means end-of-file. Conveniently, if the caller was `collect_args`, it already handles `CPP_EOF` as if it were end-of-file, and reports an error about an unterminated macro argument list.

The C standard also specifies that a new line in the middle of the arguments to a macro is treated as whitespace. This white space is important in case the macro argument is stringized. The state variable `parsing_args` is nonzero when the preprocessor is collecting the arguments to a macro call. It is set to 1 when looking for the opening parenthesis to a function-like macro, and 2 when collecting the actual arguments up to the closing parenthesis, since these two cases need to be distinguished sometimes. One such time is here: the lexer sets the `PREV_WHITE` flag of a token if it meets a new line when `parsing_args` is set to 2. It doesn't set it if it meets a new line when `parsing_args` is 1, since then code like

```
#define foo() bar
foo
baz
```

would be output with an erroneous space before `'baz'`:

```
foo
  baz
```

This is a good example of the subtlety of getting token spacing correct in the preprocessor; there are plenty of tests in the testsuite for corner cases like this.

The lexer is written to treat each of `'\r'`, `'\n'`, `'\r\n'` and `'\n\r'` as a single new line indicator. This allows it to transparently preprocess MS-DOS, Macintosh and Unix files without their needing to pass through a special filter beforehand.

We also decided to treat a backslash, either `'\'` or the trigraph `'??/'`, separated from one of the above newline indicators by non-comment whitespace only, as intending to escape the newline. It tends to be a typing mistake, and cannot reasonably be mistaken for anything else in any of the C-family grammars. Since handling it this way is not strictly conforming to the ISO standard, the library issues a warning wherever it encounters it.

Handling newlines like this is made simpler by doing it in one place only. The function `handle_newline` takes care of all newline characters, and `skip_escaped_newlines` takes care of arbitrarily long sequences of escaped newlines, deferring to `handle_newline` to handle the newlines themselves.

The most painful aspect of lexing ISO-standard C and C++ is handling trigraphs and backlash-escaped newlines. Trigraphs are processed before any interpretation of the meaning of a character is made, and unfortunately there is a trigraph representation for a backslash, so it is possible for the trigraph `'??/'` to introduce an escaped newline.

Escaped newlines are tedious because theoretically they can occur anywhere—between the `'+'` and `'='` of the `'+='` token, within the characters of an identifier, and even between the `'*'` and `'/'` that terminates a comment. Moreover, you cannot be sure there is just one—there might be an arbitrarily long sequence of them.

So, for example, the routine that lexes a number, `parse_number`, cannot assume that it can scan forwards until the first non-number character and be done with it, because this

