

Looking for a function-like macro's opening parenthesis

Function-like macros only expand when immediately followed by a parenthesis. To do this cpplib needs to temporarily disable macros and read the next token. Unfortunately, because of spacing issues (see [Token Spacing], page 15), there can be fake padding tokens in-between, and if the next real token is not a parenthesis cpplib needs to be able to back up that one token as well as retain the information in any intervening padding tokens.

Backing up more than one token when macros are involved is not permitted by cpplib, because in general it might involve issues like restoring popped contexts onto the context stack, which are too hard. Instead, searching for the parenthesis is handled by a special function, `funlike_invocation_p`, which remembers padding information as it reads tokens. If the next real token is not an opening parenthesis, it backs up that one token, and then pushes an extra context just containing the padding information if necessary.

Marking tokens ineligible for future expansion

As discussed above, cpplib needs a way of marking tokens as unexpandable. Since the tokens cpplib handles are read-only once they have been lexed, it instead makes a copy of the token and adds the flag `NO_EXPAND` to the copy.

For efficiency and to simplify memory management by avoiding having to remember to free these tokens, they are allocated as temporary tokens from the lexer's current token run (see [Lexing a line], page 5) using the function `_cpp_temp_token`. The tokens are then re-used once the current line of tokens has been read in.

This might sound unsafe. However, tokens runs are not re-used at the end of a line if it happens to be in the middle of a macro argument list, and cpplib only wants to back-up more than one lexer token in situations where no macro expansion is involved, so the optimization is safe.

Token Spacing

First, consider an issue that only concerns the stand-alone preprocessor: there needs to be a guarantee that re-reading its preprocessed output results in an identical token stream. Without taking special measures, this might not be the case because of macro substitution. For example:

```
#define PLUS +
#define EMPTY
#define f(x) =x=
+PLUS -EMPTY- PLUS+ f(=)
    ↦ + + - - + + = = =
not
    ↦ ++ -- ++ ==
```

One solution would be to simply insert a space between all adjacent tokens. However, we would like to keep space insertion to a minimum, both for aesthetic reasons and because it causes problems for people who still try to abuse the preprocessor for things like Fortran source and Makefiles.

For now, just notice that when tokens are added (or removed, as shown by the `EMPTY` example) from the original lexed token stream, we need to check for accidental token pasting. We call this *paste avoidance*. Token addition and removal can only occur because of macro expansion, but accidental pasting can occur in many places: both before and after each macro replacement, each argument replacement, and additionally each token created by the `#` and `##` operators.

Look at how the preprocessor gets whitespace output correct normally. The `cpp_token` structure contains a flags byte, and one of those flags is `PREV_WHITE`. This is flagged by the lexer, and indicates that the token was preceded by whitespace of some form other than a new line. The stand-alone preprocessor can use this flag to decide whether to insert a space between tokens in the output.

Now consider the result of the following macro expansion:

```
#define add(x, y, z) x + y +z;
sum = add (1,2, 3);
    ↦ sum = 1 + 2 +3;
```

The interesting thing here is that the tokens `'1'` and `'2'` are output with a preceding space, and `'3'` is output without a preceding space, but when lexed none of these tokens had that property. Careful consideration reveals that `'1'` gets its preceding whitespace from the space preceding `'add'` in the macro invocation, *not* replacement list. `'2'` gets its whitespace from the space preceding the parameter `'y'` in the macro replacement list, and `'3'` has no preceding space because parameter `'z'` has none in the replacement list.

Once lexed, tokens are effectively fixed and cannot be altered, since pointers to them might be held in many places, in particular by in-progress macro expansions. So instead of modifying the two tokens above, the preprocessor inserts a special token, which I call a *padding token*, into the token stream to indicate that spacing of the subsequent token is special. The preprocessor inserts padding tokens in front of every macro expansion and expanded macro argument. These point to a *source token* from which the subsequent real token should inherit its spacing. In the above example, the source tokens are `'add'` in the macro invocation, and `'y'` and `'z'` in the macro replacement list, respectively.

It is quite easy to get multiple padding tokens in a row, for example if a macro's first replacement token expands straight into another macro.

```
#define foo bar
#define bar baz
[foo]
    ↦ [baz]
```

Here, two padding tokens are generated with sources the 'foo' token between the brackets, and the 'bar' token from foo's replacement list, respectively. Clearly the first padding token is the one to use, so the output code should contain a rule that the first padding token in a sequence is the one that matters.

But what if a macro expansion is left? Adjusting the above example slightly:

```
#define foo bar
#define bar EMPTY baz
#define EMPTY
[foo] EMPTY;
    ↦ [ baz] ;
```

As shown, now there should be a space before 'baz' and the semicolon in the output.

The rules we decided above fail for 'baz': we generate three padding tokens, one per macro invocation, before the token 'baz'. We would then have it take its spacing from the first of these, which carries source token 'foo' with no leading space.

It is vital that cpplib get spacing correct in these examples since any of these macro expansions could be stringized, where spacing matters.

So, this demonstrates that not just entering macro and argument expansions, but leaving them requires special handling too. I made cpplib insert a padding token with a NULL source token when leaving macro expansions, as well as after each replaced argument in a macro's replacement list. It also inserts appropriate padding tokens on either side of tokens created by the '#' and '##' operators. I expanded the rule so that, if we see a padding token with a NULL source token, *and* that source token has no leading space, then we behave as if we have seen no padding tokens at all. A quick check shows this rule will then get the above example correct as well.

Now a relationship with paste avoidance is apparent: we have to be careful about paste avoidance in exactly the same locations we have padding tokens in order to get white space correct. This makes implementation of paste avoidance easy: wherever the stand-alone preprocessor is fixing up spacing because of padding tokens, and it turns out that no space is needed, it has to take the extra step to check that a space is not needed after all to avoid an accidental paste. The function `cpp_avoid_paste` advises whether a space is required between two consecutive tokens. To avoid excessive spacing, it tries hard to only require a space if one is likely to be necessary, but for reasons of efficiency it is slightly conservative and might recommend a space where one is not strictly needed.

Line numbering

Just which line number anyway?

There are three reasonable requirements a cpplib client might have for the line number of a token passed to it:

- The source line it was lexed on.
- The line it is output on. This can be different to the line it was lexed on if, for example, there are intervening escaped newlines or C-style comments. For example:

```
foo /* A long
comment */ bar \
baz
⇒
foo bar baz
```

- If the token results from a macro expansion, the line of the macro name, or possibly the line of the closing parenthesis in the case of function-like macro expansion.

The `cpp_token` structure contains `line` and `col` members. The lexer fills these in with the line and column of the first character of the token. Consequently, but maybe unexpectedly, a token from the replacement list of a macro expansion carries the location of the token within the `#define` directive, because cpplib expands a macro by returning pointers to the tokens in its replacement list. The current implementation of cpplib assigns tokens created from built-in macros and the `#` and `##` operators the location of the most recently lexed token. This is because they are allocated from the lexer's token runs, and because of the way the diagnostic routines infer the appropriate location to report.

The diagnostic routines in cpplib display the location of the most recently *lexed* token, unless they are passed a specific line and column to report. For diagnostics regarding tokens that arise from macro expansions, it might also be helpful for the user to see the original location in the macro definition that the token came from. Since that is exactly the information each token carries, such an enhancement could be made relatively easily in future.

The stand-alone preprocessor faces a similar problem when determining the correct line to output the token on: the position attached to a token is fairly useless if the token came from a macro expansion. All tokens on a logical line should be output on its first physical line, so the token's reported location is also wrong if it is part of a physical line other than the first.

To solve these issues, cpplib provides a callback that is generated whenever it lexes a preprocessing token that starts a new logical line other than a directive. It passes this token (which may be a `CPP_EOF` token indicating the end of the translation unit) to the callback routine, which can then use the line and column of this token to produce correct output.

Representation of line numbers

As mentioned above, cpplib stores with each token the line number that it was lexed on. In fact, this number is not the number of the line in the source file, but instead bears more resemblance to the number of the line in the translation unit.

The preprocessor maintains a monotonic increasing line count, which is incremented at every new line character (and also at the end of any buffer that does not end in a new line). Since a line number of zero is useful to indicate certain special states and conditions, this variable starts counting from one.

This variable therefore uniquely enumerates each line in the translation unit. With some simple infrastructure, it is straight forward to map from this to the original source file and line number pair, saving space whenever line number information needs to be saved. The code that implements this mapping lies in the files `line-map.cc` and `line-map.h`.

Command-line macros and assertions are implemented by pushing a buffer containing the right hand side of an equivalent `#define` or `#assert` directive. Some built-in macros are handled similarly. Since these are all processed before the first line of the main input file, it will typically have an assigned line closer to twenty than to one.

The Multiple-Include Optimization

Header files are often of the form

```
#ifndef F00
#define F00
...
#endif
```

to prevent the compiler from processing them more than once. The preprocessor notices such header files, so that if the header file appears in a subsequent `#include` directive and `F00` is defined, then it is ignored and it doesn't preprocess or even re-open the file a second time. This is referred to as the *multiple include optimization*.

Under what circumstances is such an optimization valid? If the file were included a second time, it can only be optimized away if that inclusion would result in no tokens to return, and no relevant directives to process. Therefore the current implementation imposes requirements and makes some allowances as follows:

1. There must be no tokens outside the controlling `#if-#endif` pair, but whitespace and comments are permitted.
2. There must be no directives outside the controlling directive pair, but the *null directive* (a line containing nothing other than a single `#` and possibly whitespace) is permitted.
3. The opening directive must be of the form

```
#ifndef F00
```

or

```
#if !defined F00      [equivalently, #if !defined(F00)]
```

4. In the second form above, the tokens forming the `#if` expression must have come directly from the source file—no macro expansion must have been involved. This is because macro definitions can change, and tracking whether or not a relevant change has been made is not worth the implementation cost.
5. There can be no `#else` or `#elif` directives at the outer conditional block level, because they would probably contain something of interest to a subsequent pass.

First, when pushing a new file on the buffer stack, `_stack_include_file` sets the controlling macro `mi_macro` to `NULL`, and sets `mi_valid` to `true`. This indicates that the preprocessor has not yet encountered anything that would invalidate the multiple-include optimization. As described in the next few paragraphs, these two variables having these values effectively indicates top-of-file.

When about to return a token that is not part of a directive, `_cpp_lex_token` sets `mi_valid` to `false`. This enforces the constraint that tokens outside the controlling conditional block invalidate the optimization.

The `do_if`, when appropriate, and `do_ifndef` directive handlers pass the controlling macro to the function `push_conditional`. `cpplib` maintains a stack of nested conditional blocks, and after processing every opening conditional this function pushes an `if_stack` structure onto the stack. In this structure it records the controlling macro for the block, provided there is one and we're at top-of-file (as described above). If an `#elif` or `#else` directive is encountered, the controlling macro for that block is cleared to `NULL`. Otherwise, it survives until the `#endif` closing the block, upon which `do_endif` sets `mi_valid` to `true` and stores the controlling macro in `mi_macro`.

`_cpp_handle_directive` clears `mi_valid` when processing any directive other than an opening conditional and the null directive. With this, and requiring top-of-file to record a controlling macro, and no `#else` or `#elif` for it to survive and be copied to `mi_macro` by `do_endif`, we have enforced the absence of directives outside the main conditional block for the optimization to be on.

Note that whilst we are inside the conditional block, `mi_valid` is likely to be reset to `false`, but this does not matter since the closing `#endif` restores it to `true` if appropriate.

Finally, since `_cpp_lex_direct` pops the file off the buffer stack at EOF without returning a token, if the `#endif` directive was not followed by any tokens, `mi_valid` is `true` and `_cpp_pop_file_buffer` remembers the controlling macro associated with the file. Subsequent calls to `stack_include_file` result in no buffer being pushed if the controlling macro is defined, effecting the optimization.

A quick word on how we handle the

```
#if !defined F00
```

case. `_cpp_parse_expr` and `parse_defined` take steps to see whether the three stages ‘!’, ‘defined-expression’ and ‘end-of-directive’ occur in order in a `#if` expression. If so, they return the guard macro to `do_if` in the variable `mi_ind_macro`, and otherwise set it to `NULL`. `enter_macro_context` sets `mi_valid` to `false`, so if a macro was expanded whilst parsing any part of the expression, then the top-of-file test in `push_conditional` fails and the optimization is turned off.

File Handling

Fairly obviously, the file handling code of `cpplib` resides in the file `files.cc`. It takes care of the details of file searching, opening, reading and caching, for both the main source file and all the headers it recursively includes.

The basic strategy is to minimize the number of system calls. On many systems, the basic `open ()` and `fstat ()` system calls can be quite expensive. For every `#include`-d file, we need to try all the directories in the search path until we find a match. Some projects, such as `glibc`, pass twenty or thirty include paths on the command line, so this can rapidly become time consuming.

For a header file we have not encountered before we have little choice but to do this. However, it is often the case that the same headers are repeatedly included, and in these cases we try to avoid repeating the filesystem queries whilst searching for the correct file.

For each file we try to open, we store the constructed path in a splay tree. This path first undergoes simplification by the function `_cpp_simplify_pathname`. For example, `/usr/include/bits/./foo.h` is simplified to `/usr/include/foo.h` before we enter it in the splay tree and try to `open ()` the file. CPP will then find subsequent uses of `foo.h`, even as `/usr/include/foo.h`, in the splay tree and save system calls.

Further, it is likely the file contents have also been cached, saving a `read ()` system call. We don't bother caching the contents of header files that are re-inclusion protected, and whose re-inclusion macro is defined when we leave the header file for the first time. If the host supports it, we try to map suitably large files into memory, rather than reading them in directly.

The include paths are internally stored on a null-terminated singly-linked list, starting with the `"header.h"` directory search chain, which then links into the `<header.h>` directory chain.

Files included with the `<foo.h>` syntax start the lookup directly in the second half of this chain. However, files included with the `"foo.h"` syntax start at the beginning of the chain, but with one extra directory prepended. This is the directory of the current file; the one containing the `#include` directive. Prepending this directory on a per-file basis is handled by the function `search_from`.

Note that a header included with a directory component, such as `#include "mydir/foo.h"` and opened as `/usr/local/include/mydir/foo.h`, will have the complete path minus the basename `'foo.h'` as the current directory.

Enough information is stored in the splay tree that CPP can immediately tell whether it can skip the header file because of the multiple include optimization, whether the file didn't exist or couldn't be opened for some reason, or whether the header was flagged not to be re-used, as it is with the obsolete `#import` directive.

For the benefit of MS-DOS filesystems with an 8.3 filename limitation, CPP offers the ability to treat various include file names as aliases for the real header files with shorter names. The map from one to the other is found in a special file called `'header.gcc'`, stored in the command line (or system) include directories to which the mapping applies. This may be higher up the directory tree than the full path to the file minus the base name.

Concept Index

A

assertions 9

C

controlling macros 19

E

escaped newlines 3

F

files 21

G

guard macros 19

H

hash table 9

header files 1

I

identifiers 9

interface 1

L

lexer 3

line numbers 17

M

macro expansion 11

macro representation (internal) 11

macros 9

multiple-include optimization 19

N

named operators 9

newlines 3

P

paste avoidance 15

S

spacing 15

T

token run 5

token spacing 15