Recursive macros

Document number:	p???
Date:	2023-10-19
Project:	Programming language C++
Audience:	EWG
Author:	Kelbon
Reply-to:	Kelbon kelbonage@gmail.com

Contents

1 Abstract

We propose a special __THIS_MACRO__ identifier, which makes it possible to create recursive macros.

2 Motivation

There are huge amount of code, which uses code generation, fragile and confusing macros or misuse templates for code generation

- nlohmann json[link]
- P99[link]
- Metalang99[link]
- Boost.Preprocessor[link]
- generated code in boost::pfr[link]
- generated code in AnyAny[link]

Such code is difficult to maintain, test, write and debug. It's easy to make a mistake and compiler is often unable to produce a good error.

But people need these opportunities.

Existing solutions do not allow programmers to avoid the problems described above, and code generation adds its own: once a code generation script has been written, it requires support in order to be able to change the generated code in the future.

Modern C++ provides opportunities to get rid of the preprocessor where it was never needed, but there will always be places where the preprocessor is a best reflection of intentions.

Also C++20 adds __VA_OPT__, which is ideal for recursive macros, but there are no such thing in C++!

3 Proposed solution

It is proposed to add a special __THIS_MACRO__ token, which is behaves as macro name in macro definition, except name of the macro containing the __THIS_MACRO__ token in the definition replaced in [cpp.rescan] even if it is found during scan of the replacement list

To handle infinite recursion, there are implementation defined recursion depth limit, like in other similar situations.

This makes possible to write such macros:

```
#define fold_right(op, head, ...) \
( head __VA_OPT__(op __THIS_MACRO__(op, __VA_ARGS__)) )
/* expands to (1 + (2 + (3))) */
int i = fold_right(+, 1, 2, 3);
```

Calculate count of tokens

```
#define TOKCOUNT_IMPL(head, ...) \
1 __VA_OPT__(+ __THIS_MACRO__(__VA_ARGS__))
/* works for zero args too */
#define tokcount(...) \
(0 __VA_OPT__(+ TOKCOUNT_IMPL(__VA_ARGS__)) )

/* expands to (0 + 1 + 1 + 1 ) */
tokcount(1, A, "gdfdg")
```

Replace for boost::pfr code generation script:

```

#define try_expand(value, head, ...)            \
if constexpr (aggregate_size<decltype(value)>() \
              == tokcount(+1, __VA_ARGS__)) {   \
  auto [head __VA_OPT__(,) __VA_ARGS__] = value;\
  return tie(head __VA_OPT__(,) __VA_ARGS__);   \
}                                               \
__VA_OPT__(__THIS_MACRO__(value, __VA_ARGS__))

constexpr auto tie_aggregate(auto&& aggregate) {
  /* expands to similar functional
     as boost::pfr for up to 3 members */
  try_expand(aggregate, _3, _2, _1);
}
```

4 Proposed wording

[cpp.replace.general]

> The identifiers __THIS_MACRO__, __VA_ARGS__ and __VA_OPT__ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters. Token __THIS_MACRO__ behaves as if it is replaced by name of macro in which definition it appears, except name of such macro always replaced during [cpp.rescan]

[cpp.rescan]

> If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens) and its definition does not contain __THIS_MACRO__ token, it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

[implimits]

> Recursively nested macro name expansion[2048]

5 Implementation experience

The implementation in clang and examples, could be found here[link]