Recursive macros with C++20 `__VA_OPT__`

David Mazières

June, 2021

Introduction

Wouldn't it be nice if you could define pretty-printable enums in C++? Well, in C++20, you can define a macro that both creates an enum type and defines a function converting the enum values to strings. Here's an example of such a macro in action:

```
MAKE_ENUM(MyType, ZERO, ONE, TWO, THREE);

void
test(MyType e)
{
  std::cout << to_cstring(e) << " = " << e << std::endl;
}

int
main()
{
  test(ZERO);
  test(ONE);
  test(TWO);
  test(THREE);
}
```

Output:

```
ZERO = 0
ONE = 1
TWO = 2
THREE = 3
```

The key to making this work is a new pre-processor feature in C++20, `__VA_OPT__(x)`, which expands to `x` when a variable-argument macro has more than zero arguments and to nothing otherwise. This is exactly what you need to implement the base case in recursive macros, allowing things like a `FOR_EACH` macro that applies another macro to each of its arguments. Here's how you can define `MAKE_ENUM` in terms of `FOR_EACH`:

```
#define ENUM_CASE(name) case name: return #name;

#define MAKE_ENUM(type, ...)                    \
enum type {                                     \
  __VA_ARGS__                                   \
};                                              \
constexpr const char *                          \
to_cstring(type _e)                             \
{                                               \
  using enum type;                              \
  switch (_e) {                                 \
  FOR_EACH(ENUM_CASE, __VA_ARGS__)              \
  default:                                      \
    return "unknown";                           \
  }                                             \
}
```

The full code is available in make_enum.cc. The rest of this blog post explains how to define `FOR_EACH` in a simple and general way. Though there's an (arbitrary) limit to the size of the argument list, the limit is exponential in the length of the source code. With an extra 5 lines of code, you can have over 300 arguments, which seems like plenty and is way more acceptable than old approaches requiring a separate macro for each possible number of arguments.

So in this blog post, I'll attempt to explain how C/C++ macros actually work and then show how to combine that with C++20 `__VA_OPT__` to do some cool things.

C macro overview

C and C++ support two kinds of macros, *object-like* macros, which have no arguments, and *function-like* macros, which require arguments. Here are some simple examples.

```
#define OL 123            // object-like macro
#define FL(x) ((x)+1)     // function-like macro
```

The body of a macro—i.e., the part after the macro and optional arguments—is known as the *substitution list*. Above, the substitution list is the single token `123` for `OL`, and the token list `((x)+1)` for `FL`.

Macro expansion occurs after the C/C++ preprocessor, *cpp*, has turned the program source code into a series of lexical *tokens*. Identifiers such as `FL`, numbers, character literals, quoted strings, parentheses, and operators such as `+` are all examples of tokens. Cpp effectively transforms a list of input tokens to output tokens by copying many tokens as is, but expanding macros in appropriate places.

Cpp was designed to guarantee termination of source code preprocessing. Personally, I think termination is way overrated in programming languages. I mean, what comfort is the fact that C is decidable when a trivial cpptorture.c program can require over 100 years and many

exabytes of memory to compile? But I digress. More practically, people like to write code like this, from the linux `<sys/epoll.h>` header:

```
enum EPOLL_EVENTS
  {
    EPOLLIN = 0x001,
#define EPOLLIN EPOLLIN
    EPOLLPRI = 0x002,
#define EPOLLPRI EPOLLPRI
    EPOLLOUT = 0x004,
#define EPOLLOUT EPOLLOUT
  /* ... */
  };
```

It's nice to make `EPOLL_EVENTS` an enum, since doing so aids in debugging and is more elegant. But it's also nice for programs to be able to check for the availability of a particular flag with `#ifdef EPOLLPRI`. So the `<sys/epoll.h>` header solves both problems, taking advantage of the fact that cpp mostly doesn't expand macros recursively. After these definitions, the token `EPOLLIN` will expand to itself once and then stop expanding, so it's effectively equivalent to an enum that also supports `#ifdef`.

To prevent recursion, cpp associates a bit with every macro that has been defined. The bit reflects whether the macro is currently being replaced with its substitution list, so let's call it the *replacing* bit. Cpp furthermore associates a bit with each token in the input stream, signifying that the token can never be macro-expanded. Let's call the latter bit the *unavailable* bit. Initially, the replacing and unavailable bits are all clear.

As cpp processes each input token `T`, it sets `T`'s unavailable bit and decides whether or not to macro-expand `T` as follows:

1. If `T` is the name of a macro for which the *replacing* bit is true, cpp sets the *unavailable* bit on token `T`. Note that even if `T` is not in a context where it could be macro-expanded—because it's a function-like macro not followed by "`(`"—cpp still sets the unavailable bit. Moreover, once the unavailable has been set on an input token, it is never be cleared.

2. If `T` is the name of an object-like macro and `T`'s *unavailable* bit is clear, then `T` is expanded.

3. If `T` is the name of a function-like macro, `T`'s *unavailable* bit is clear, and `T` is followed by `(`, then `T` is expanded. Note, however, that if `T` is called with an invalid number of arguments, then the program is ill-formed.

If cpp decides not to macro-expand `T`, it simply adds `T` to the current output token list. Otherwise, it expands `T` in two phases.

1. When `T` is a function-like macro, cpp scans all of the arguments supplied to `T` and performs macro expansion on them. It scans arguments the same as normal token processing, but instead of placing output tokens in the main preprocessor output, it

builds a replacement token list for each of `T`'s arguments. It also remembers the original, non-macro-expanded arguments for use with `#` and `##`.

2. Cpp takes `T`'s substitution list and, if `T` had arguments, replaces any occurrences of parameter names with the corresponding argument token lists computed in step 1. It also performs stringification and pasting as indicated by `#` and `##` in the substitution list. It then logically prepends the resulting tokens to the input list. Finally, cpp sets the replacing bit to true on the macro named `T`.

   With the replacing bit true, cpp continues processing input as usual from the tokens it just added to the input list. This may result in more macro expansions, so is sometimes called the *rescan phase*. Once cpp has consumed all tokens generated from the substitution list, it clears the replacing bit on the macro named `T`.

Let's look at a simple example:

```
FL(FL(5))     // => ((((5)+1))+1)
```

In phase 1, the argument of the outer macro, namely "`FL(5)`," gets expanded to the token list `((5)+1)`, yielding `FL(((5)+1))`. Expanding the outer `FL` macro substitutes this argument for the parameter `x` in the substitution list, producing `((((5)+1))+1)`. The result should be reasonably intuitive. The one thing to note is that because expansion of the inner `FL` happened in phase 1, `FL`'s replacing bit was clear and no tokens ever needed their unavailable bits set.

Now let's look at a more interesting example:

```
#define ID(arg) arg
ID(ID)(ID)(X)    // => ID(ID)(X)
```

Consider the first part of the token sequence, namely `ID(ID)`. We start in phase 1 by macro-expanding the inner `ID`, but since it's a function-like macro not followed by `(`, cpp decides not to expand it. Hence, cpp replaces `arg` with `ID` in the outer `ID`'s substitution list, and pushes the result onto the input list. Then it sets macro `ID`'s replacing bit and proceeds to phase 2 (rescan). Upon processing the first token, `ID`, cpp will set its unavailable bit (since `ID` has replacing true) and not expand it. Finally, cpp will clear `ID`'s replacing bit, but at this point there is nothing left to expand because the third `ID` is not followed by `(`.

It turns out there's actually a known ambiguity in the specification as to when exactly the replacing bit gets cleared. What happens if a macro expansion ends with a function-like macro, but the arguments to that macro include tokens from after the expansion? In practice, compilers seem to do the intuitive thing and clear the replacing bit exactly before the first token that entirely follows the macro expansion. For example:

```
#define LPAREN (
#define ID2(arg) arg

ID(ID2)(ID)(X)          // => X
ID(ID2 LPAREN)ID)(X)    // => X
```

```
ID(ID2 LPAREN ID))(X) // => ID(X)
ID(ID2 (ID))(X)      // => ID(X)
```

In the above examples, various portions of the `(ID)` argument to `ID2` are moved into the argument of the first `ID`, and you can see that the second `ID` starts getting the unavailable bit set as soon as it moves into the substitution list of the first `ID`.

Recursive macros

Of course, the C preprocessor trivially implements recursion via the `#include` directive. Files can include themselves and appropriately `#undef`/`#define` constants to implement the base case with `#if` conditionals. Include-file recursion isn't all that practical, but it turns out you can also expand macros recursively, or at least mutually recursively. The trick, which I first saw proposed by Paul Fultz, is to avoid setting the unavailable bit on the macro that you want to expand recursively by hiding the token until another macro's rescan phase.

Let's look at an example:

```
#define ID(arg) arg
#define PARENS () // Note space before (), so object-like macro
#define F_AGAIN() F
#define F() f F_AGAIN PARENS()

F()          // => f F_AGAIN ()()
ID(F())      // => f f F_AGAIN ()()
ID(ID(F())) // => f f f F_AGAIN ()()
```

When we expand `F()`, note that `F_AGAIN` is not followed by `(`, so it does not get expanded as a macro. Sure, one step later, `PARENS` gets expanded to `()`, but at this point cpp has already output the token `F_AGAIN`, so it's too late to decide to expand it. Hence, the output of `F()`—namely `f F_AGAIN ()()`—may contain an unexpanded macro call, but the unavailable bits are clear on all tokens.

Now consider what happens when we call `ID(F())`. Well, fist we expand the argument `F()` to `f F_AGAIN ()()`. Then we are done, so we clear `F`'s replacing bit. Next, `ID` substitutes `f F_AGAIN ()()` for `arg` in its substitution list (namely the single token `arg`). So the preprocessor sets `ID`'s replacing bit and rescans `f F_AGAIN ()()`, causing `F_AGAIN` and then `F` to expand. But of course the same `PARENS` trick prevents the second `F_AGAIN` from getting expanded.

Each time we pass `F()` through our identity macro `ID`, it gets expanded one more time. So we can't recurse indefinitely, but we can set an arbitrarily high maximum number of times. And since we can easily generate a number of macro calls exponential in the number of lines of code we write (remember our trivial cpptorture.c program?), the real limit is how much time and memory we have for cpp, not the fact that cpp isn't turing complete. Here are 5 lines of code that re-scan macros 342 times (`EXPAND4` gets called 256 times, but the intermediary macros cause rescan as well):

```
#define EXPAND(arg) EXPAND1(EXPAND1(EXPAND1(EXPAND1(arg))))
#define EXPAND1(arg) EXPAND2(EXPAND2(EXPAND2(EXPAND2(arg))))
#define EXPAND2(arg) EXPAND3(EXPAND3(EXPAND3(EXPAND3(arg))))
#define EXPAND3(arg) EXPAND4(EXPAND4(EXPAND4(EXPAND4(arg))))
#define EXPAND4(arg) arg
```

Variable-argument macros

C++11 added variable-argument macros. When the last macro parameter in a `#define` is `...` rather than an identifier, it can accept an arbitrary number of arguments and the special token `__VA_ARGS__` in the substitution list expands to all of these arguments, separated by commas. The canonical example is:

```
#define LOG(...) printf(__VA_ARGS__)
```

Unfortunately, there's a slight annoyance that, in many situations, it's hard to write a macro that generates syntactically correct C code when `...` represents zero arguments. For example, suppose you want a macro that prints messages in brackets. You might try to do something like this:

```
#define LOG(fmt, ...) printf("[" fmt "]", __VA_ARGS__)

LOG("level %d", lvl); // => printf("[" "level %d" "]", lvl);
```

Here we take advantage of the fact that C concatenates adjacent string constants. Since the first argument to LOG, corresponding to parameter `fmt`, is expected to be a string constant, we can construct a new format argument to `printf` in which this string has been bracketed. Unfortunately, this doesn't work if there are no arguments after the format string:

```
LOG("hello");          // => printf("[" "hello" "]", );
```

The extra comma in `printf("hello",)` is a syntax error in C and C++. C++20 solved this problem by adding a new special identifier, `__VA_OPT__`. The sequence `__VA_OPT__(x)`, which is only legal in the substitution list of a variable-argument macro, expands to `x` if `__VA_ARGS__` is non-empty and to nothing if it is empty. That allows us to fix the `LOG` macro by suppressing the comma when the argument list is empty:

```
#define LOG(fmt, ...) printf("[" fmt "]" __VA_OPT__(,) __VA_ARGS__)
LOG("hello");          // => printf("[" "hello" "]" );
```

But of course, as an unintended benefit, differentiating between empty and non-empty argument lists is exactly the mechanism we need to implement the base case in recursion...

The `FOR_EACH` macro

We now have all the pieces we need to implement a `FOR_EACH` macro:

```
#define PARENS ()

#define EXPAND(...) EXPAND4(EXPAND4(EXPAND4(EXPAND4(__VA_ARGS__))))
#define EXPAND4(...) EXPAND3(EXPAND3(EXPAND3(EXPAND3(__VA_ARGS__))))
#define EXPAND3(...) EXPAND2(EXPAND2(EXPAND2(EXPAND2(__VA_ARGS__))))
#define EXPAND2(...) EXPAND1(EXPAND1(EXPAND1(EXPAND1(__VA_ARGS__))))
#define EXPAND1(...) __VA_ARGS__

#define FOR_EACH(macro, ...)                                    \
  __VA_OPT__(EXPAND(FOR_EACH_HELPER(macro, __VA_ARGS__)))
#define FOR_EACH_HELPER(macro, a1, ...)                         \
  macro(a1)                                                     \
  __VA_OPT__(FOR_EACH_AGAIN PARENS (macro, __VA_ARGS__))
#define FOR_EACH_AGAIN() FOR_EACH_HELPER

FOR_EACH(F, a, b, c, 1, 2, 3)   // => F(a) F(b) F(c) F(1) F(2) F(3)
```

Note that we've tweaked `EXPAND` so that it handles macros that output commas by simply using `__VA_ARGS__` instead of a named `arg`.

The bulk of the work happens in `FOR_EACH_HELPER(macro, a1, ...)`, which applies `macro` to argument `a1`, and then uses `__VA_OPT__` to recurse if the remaining arguments are not empty. Just as in the previous section, it uses the `PARENS` trick to enable recursion. The only catch, of course, is that we have to keep re-scanning the macro, which is why the `FOR_EACH` macro wraps `FOR_EACH_HELPER` in the `EXPAND` macro we saw before. For good measure, `FOR_EACH` also uses `__VA_OPT__` to handle the case of an empty argument list.

Will I use this in production code? I'm thinking about it. In my first decade of C++ programming, I used to think that being a good C++ programmer was all about showing how clever you are. Now as a wise old senior faculty member, I know that being a good C++ programmer is all about showing restraint. You need to know both *how* to be clever and *when* to be clever. So let's do the cost-benefit analysis, starting with the alternative approaches:

1. You could manually maintain separate `enum` declarations and pretty-printer/scanner functions, with the risk that they could get out of sink.

2. You could generate the C++ code using another program, but this complicates the build process and typically doesn't make the code any more readable. C++ isn't a great language for generating text, and if you use perl or python or bash, the code won't necessarily be more transparent to other C++ programmers.

3. What I'm currently doing: my `MAKE_ENUM` equivalent macro stringifies `__VA_ARGS__` and passes it to a ~25-line function that parses it into a `std::vector<std::string>`. I then have an `EnumTab` type containing `std::map`s to get between enum values and strings. The `EnumTab` constructor takes a vector of strings and a `std::initializer_list` of constants. So basically my macro ends up generating a function like this for every enum type:

```
static inline const EnumTab &
getEnumTab(const Enum *)
{
  static const EnumTab tab(EnumTab::parse_va_args(#__VA_ARGS__),
                           {__VA_ARGS__});
  return tab;
}
```

This is unreleased code, and the grossness around enum parsing (these enums need to be read and written to human-readable files) is one of the reasons I don't want to show it publicly yet.

So I think the `FOR_EACH` approach is actually a net win over options 2 and 3. The most restrained option, which you should always be considering in C++, is number 1.

How much are we paying in complexity for the use of `FOR_EACH`? It's definitely tricky to understand how `FOR_EACH` works if you don't know how cpp works. It's also, unfortunately, hard to figure out how cpp works. I was unable to understand the C++ language specification for macro replacement until I'd already understood how cpp works. https://en.cppreference.com/ doesn't get into nearly the level of detail necessary. On the other hand, I now have this blog post I can reference in my source code, so writing this post is actually part of deciding whether or not I want to use the trick. Of course, others should feel free to do the same... I place all the cpp macros in this blog post in the public domain.

`FOR_EACH` is also far from the grossest use of macros I've seen. It doesn't even use token pasting (`##`) to synthesize new tokens that you can't textually search for. Even though the implementation is tricky to understand, it's at least short. More importantly, the interface to `FOR_EACH` is quite intuitive. For a multi-line C macro, I think `MAKE_ENUM` is fairly readable. And once you employ `FOR_EACH` in one place, you can potentially amortize the complexity over other uses of the macro.

Whatever you think of the trade-offs, this much is certain: the introduction of `__VA_OPT__` makes `FOR_EACH` decidedly more palatable than the brittle and disgusting approaches with older versions of C++, to the point that it's at least worth seriously considering.