Macros And Standard Library Modules

`import` should suffice

Document #:	P2654R0
Date:	2023-05-15
Project:	Programming Language C++
Audience:	Evolution Incubator
Revises:	N/A

Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

C++23 introduces the notion of library modules that export the whole content of the C++ Standard Library, except for any parts that are defined as macros. This paper reviews the library macros that are therefore not exported, and looks for ways to export that same functionality without requiring the modules language feature to become aware of macros.

2 Revision history

2.1 R0: Varna 2023

Initial draft of the paper.

3 Introduction

Modules, introduced in C++20, are the preferred way to export and import libraries in modern C++. They offer benefits to efficient translation of C++ source code, and isolation against unexpected use of macros. As such, they do not allow for exporting of macros, so libraries with macro interfaces are not directly supported, which is most typically (but not exclusively) an issue for libraries interoperating with C.

When a library interface does require use of macros, the next best alternative is for consumers to import a header module. In this mode, the compiler tries to parse a header as a module interface, but also allows macros to escape as-if it were `#include`-ed. Not all headers are suitable for use as header modules, and in particular, while the standard C++ library headers are mandated to be compatible, there is no such guarantee for the C standard library headers nor for the C compatibility headers such as `<cstddef>`.

Finally, where a header is not an importable header module, the user must fall back on the traditional preprocessor `#include`. The C++ library requirements ensure that the same functionality can be both imported and `#include`-ed as long as there is no devious use of the preprocessor to cause their behavior to differ. Note that such abuse is highly irregular and unlikely to be encountered in practice.

This paper observes that several parts of the C language are exposed as macros, rather than straight language facilities, and so are not usable in a modern C++ code base without resorting all the way back to the direct use of `#include`. It explores some alternative designs that could remove this limitation.

4 Scoping the problem

There are a number of language and library facilities expressed through macros in the C++23 standard. Here we provide a quick overview of each, and delegate deeper analysis to a separate paper for each case, where a variety of resolutions are proposed.

4.1 Literal values

The most commonly encountered macros in the standard are amacros substituting for literal values. In many cases, that is so these macros can be evaluated during preprocessor logic, such as `#if` directives, so they are not easily replaced by `constexpr` literals in C++. Likewise, macros corresponding to string literals must continue to support string literal concatenation, which would not be the case for a `const char *` literal.

The resolution proposed by (paper pending) will be a new preprocessor directive that does not do text replacement, and is suitable for use in module interfaces.

4.2 `assert`

The second most commonly encountered macro is likely to be `assert`. This macro already suffers a variety of problems in C++ due to C++ having a larger set of balanced bracket tokens than C, leading to problems with comma-separated lists, e.g., template arguments.

Paper [P2884R0] takes a close look at this macro, and whether it could be defined as an operator in C++26 by taken `assert` as a reserved word. There are a variety of concerns pursuing this direction that are addressed in that paper.

In a separate line of development, SG21 are working on a contracts facility for C++26 that will include an improved assertion facility. The syntax to express assertions, and contract checks in general, will be the main topic for SG21 following the Varna meeting, so it would be good to know early if there is any likelihood of adopting `assert` as a keyword token that would be available to them. Note that SG21 has not requested this token, the observation arises only as I am preparing this set of papers.

4.3 `offsetof`

See paper [P2883R0].

4.4 setjmp

The `setjmp/longjmp` facility interacts directly with the C++ object model and the notion of object lifetimes. It really should be adopted into the core language, turning the parts expressed through macros today into keywords.

4.5 va_arg and friends

This is a fundaC++.mental language feature, so should be accessible through `import`. Suggests defining the macro behavior with keywords in

4.6 errno

This is a tricky one, no ideas for progress yet.

4.7 ATOMIC_scalar_LOCK_FREE

These macros should be predefined by the compiler as part of the platform support.

4.8 ATOMIC_FLAG_INIT

Resolve by removing macro when adopting the C23 library that has already removed it.

5 Tracking Paper Progress

Here we provide a checklist to track to progress of the separate papers that will resolve each of the specific concerns of this larger paper.

Feature	By Paper	Owner
`offsetof` macro	[P2883R0]	EWGI
`assert` macro	[P2884R0]	EWGI
Macros as literal values	Pending	EWGI
Accessing variadic function arguments	Pending	EWGI
Enabling `longjmp`	Pending	EWGI
`errno` is a macro	Pending	EWGI

6 Acknowledgements

Thanks to Michael Parks for the pandoc-based framework used to transform this document's source from Markdown.

7 References

[P2883R0] Alisdair Meredith. 2023-05-15. `offsetof` Should Be A Keyword In C++26.
https://wg21.link/p2883r0

[P2884R0] Alisdair Meredith. 2023-05-15. `assert` Should Be A Keyword In C++26.
https://wg21.link/p2884r0