Macros And Standard Library Modules

`import` should suffice

Document #:	P2654R1
Date:	2025-01-13
Project:	Programming Language C++
Audience:	Evolution Incubator
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

R1: 2025 January mailing (pre-Hagenberg)

— Fixed reported typos
— Removed revision history's redundant subsection numbering
— Corrected that `ATOMIC_VAR_INIT` is removed in C23, not `ATOMIC_FLAG_INIT`
— Better described the proposals for `offsetof` and var-args functions
— Concluded that `<cassert>` should be obviated my `contract_assert`
— Updated the paper tracker
— Added references to work in [P3208R0] on C stream objects
— Added references to work in [P3550R0] on C-style variadic functions
— Reorganized the paper around the categories of macro

R0: 2023 May mailing (pre-Varna)

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 macros 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.

4.1.1 Literal headers

Several standard library headers comprise only macros that provide literal values. These headers should be added to the set of importable header units.

The list of such headers is:

— `<climits>`
— `<cfloat>`
— `<version>`

Close, and for consideration:

— `<cerrno>` — can we safely import the `errno` macro?

4.1.2 `<version>`

This header is importable, and an easy way to gain access to the feature macros in the standard library. However, it is distracting to have to import a second header to access library functionality that would have been present including or importing the equivalent standard headers — that is where a future literal values preprocessor feature would help.

4.1.3 `ATOMIC_scalar_LOCK_FREE`

The nature of atomic operations is a fundamental property of the specific implementation of the abstract machine that a program is compiler to execute on. As such,these macros should be predefined by the compiler as part of its platform support, rather than left to the library.

4.1.4 Common C Library macros

Some C Library macros are close to ubiquitous, and appear in multiple C headers. It might be useful to extract those common definitions into another importable header unit, comprising nothing but the common macros.

— `NULL`
— `EOF`

4.2 Non-literal object-like macros

There are a variety of object-like macros in the C and C++ standard libraries that are more than literal values, creating a bigger problem to export their functionality.

One general approach that might address all of the specific concerns below is to have an importable header that provides nothing but these macro definitions, allowing an import that does not drag duplicate declarations from the Standard Library into the global module fragment. Such a header could be considered an analog of `<version>`, for another category of macros.

4.2.1 `errno`

Many C standard library function report failure through the `errno` macro, defined in the header `<cerrno>` along with macros for a number of error codes.

Could we declare `<cerrno>` an importable header, or is there non-importable deep magic in the implementation of `errno`?

4.2.2 C streams

See [P3208R0].

4.3 Function-like macros

Function-like macros provide important functionality, often in support of core language features. Where providing core language support, we might consider promoting the macro to keywords and directly supporting these features in the language.

In other cases, macros are here to support common source in C headers, and as such may not be a concern for a modular C++ that does not directly include such headers.

4.3.1 `offsetof`

The `offsetof` macro has to solve some interesting problems, such as how to describe the name of a data member — something that is not otherwise expressible in the language. In order to properly handle the complexities of C++ in addition to the needs of C, we recommend adopting `offsetof` as a keyword in C++. See paper [P2883R0] for more details.

4.3.2 `setjmp`

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

4.3.3 `va_arg` and friends

Paper [P3550R0] will explores the concerns more completely. In particular, C-style variadic functions have been highlighted as a security concern, so that paper explores the direction of removing such functionality from C++ (but not C) rather than better enabling that feature in modules.

4.3.4 `ATOMIC_FLAG_INIT`

This macro is not needed in C++, as its purpose is a duplicate of the constexpr default constructor. We believe it is provided only for header portability with C code, and as such is not relevant to modules, that are not concerned with writing C compatible code — at least until C implements its own compatible modules feature.

4.3.5 `ATOMIC_VAR_INIT`

The macro `ATOMIC_VAR_INIT` was initially provided as a header-compatible syntax with C to initialize atomic variables. It is no longer needed by either language; it has been removed from C23; paper [P3366R0] proposes removing this deprecated macro from C++.

4.3.6 `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] tool a close look at this macro, and whether it could be defined as an operator in C++26 by taking `assert` as a reserved word. There are a variety of concerns pursuing this direction that were addressed in that paper, but ultimately rejected by EWG.

In a separate line of development, SG21 are working on a contracts facility for C++26 that will include an improved assertion facility. The direction going forward is that assertion facilities in C++ will be better expressed using `contract_assert` statements, and there are already papers examining how best to integrate contract assertions with the C macro.

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]	REJECTED
Variadic function support	[P3550R0]	EWGI
Enabling `longjmp`	Pending	EWGI
Macros as literal values	Pending	EWGI
`errno` is a macro	Unknown	EWGI
C streams	[P3208R0]	EWGI
`ATOMIC_scalar_LOCK_FREE`	Pending	SG1
`ATOMIC_FLAG_INIT`	No action	No action
`ATOMIC_VAR_INIT`	[P3366R0]	SG1

6 Acknowledgements

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

Thanks to Bryce for the spell-check!

7 References

[P2883R0] Alisdair Meredith. 2023-05-19. 'offsetof' Should Be A Keyword In C++26.
https://wg21.link/p2883r0

[P2884R0] Alisdair Meredith. 2023-05-19. 'assert' Should Be A Keyword In C++26.
https://wg21.link/p2884r0

[P3208R0] Sunghyun Min. 2024-04-16. import std; and stream macros.
https://wg21.link/p3208r0

[P3366R0] Alisdair Meredith. 2024-08-15. Remove Deprecated Atomic Initialization API from C++26.
https://wg21.link/p3366r0

[P3550R0] Alisdair Meredith. 2025-01-13. Imports Cannot ….
https://wg21.link/d3550r0