`offsetof` Should Be A Keyword In C++26

Supporting standard C++23 macros in module `std`

Document #:	P2883R0
Date:	2023-05-12\5
Project:	Programming Language C++
Audience:	Evolution Incubator
Revises:	N/A

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

Contents

1 Abstract

Macros cannot be exported from a C++ module. This proposal suggests that the C++23 Standard Library macro `offsetof` would be better specified as a keyword, removing it from the set of library features that are *not* made available by importing the standard library module `std`.

2 Revision history

2.1 R0: Varna 2023

Initial draft of the paper.

3 Introduction

C++23 introduced the standard library module `std` that is intended to import the whole standard library, see [P2465R3]. However, this module leaves a gap for all the library facilities that are specified as macros. Paper [P28654R0] is tracking progress of a set of papers that attempt to support all language facilities that are currently specified with the aid of macros with that single import. This paper addresses the macro `offsetof`.

One key question is whether `offsetof` is a significant enough feature to be investing committee time in. I believe that this macro makes one of the stronger cases for an improved C++ experience by integrating it into the core language and resolving outstanding issues. As a first paper on moving standard library macros into the language, it is a good gauge of how much we want to invest resources into such a project.

4 Stating the problem

There are a few issues with the `offsetof` macro that we would like to address, primarily motivated to start the word due to the poor interaction with `import std;`.

4.1 Macros cannot be exported

The macro `offsetof` is defined in the `<cstddef>` header, which is not an importable header unit. While the contents of this header are exported from the standard library module `std`, such exports cannot include macros, as the grammar for module interface units is defined only in terms of declarations with external linkage, 10.2 [module.interface], which cannot describe a macro. Likewise, the standard library module `std.compat` cannot export macros either.

If `<cstddef>` were an importable header unit, then macros would be available by importing that header module 15.5 [cpp.import], but as `<cstddef>` is not an importable header unit, users must directly `#include` this header to access the `offsetof` macro. This does not play well with a long term goal of moving C++ past the preprocessor as part of how we build code.

4.2 Macros mangle C++

As specific macros are not direct language features, but rather manipulate source text, there are often surprising outcomes when the C++ source code to be supplied as an argument to a macro includes the `,` character; the macro subsystem will see the comma, and treat it as a separator between macro arguments. This problem with commas often arises with template arguments and brace-initalizers, but does not occur in the common case of regular function arguments due to the nested brackets, e.g., `MACRO(call(1,2))`.

The common workaround for such issues is to wrap each macro argument in its own pair of brackets. However, this does not work for the `offsetof` macro as neither argument supports parentheses.

Also, it is not clear whether a type expression, such as produced by the `decltype` operator, is valid for the first argument or whether you must literally spell the name of a type as the token used preprocessing the macro. In practice, it appears to work.

4.2.1 Simple Example

```
import std;
#include <cstddef>  // not an imoportable heder unit
```

```
template <typename A, typename B>
struct Test {
    int data;
};

using TestInts = Test<int, int>;
static_assert(offsetof( TestInts,        data) == 0);  // OK

~~static_assert(offsetof( Test< int,int>, data) == 0); // error~~
~~static_assert(offsetof((Test< int,int>), data) == 0); // error~~

#define WRAP(...) __VA_ARGS__
static_assert(offsetof( WRAP(Test<int, int>), data) == 0);  // OK?  Assumes no nested macros in implement
#undef  WRAP


static_assert(offsetof( decltype(Test<int, int>{}), data) == 0);  // OK?

// General case that does not assume default constructability
using namespace std;
static_assert(offsetof( remove_reference_t<
                                          decltype( declval<Test<int, int>>() )
                                          >, data) == 0);
```

4.3 Unnecessary undefined behavior

The valid set of arguments to the `offsetof` macro is restricted. It is conditionally supported for the first type argument to be type that is not a standard layout class type; note that "conditionally supported" requires non-supported cases to be a diagnosable error, so there should be no UB here.

However, for the second argument the standard says:

> The result of applying the `offsetof` macro to a static data member or a function member is undefined.

This is also a statically determined property at the point the macro is called, so should also be a diagnosable error in C++. This situation does not arise in C.

4.4 Result cannot be used in pointer arithmetic

The result of the `offsetof` macro cannot be used with regular pointer arithmetic to produce a pointer with the address of a non-static data member, as pointer arithmetic on the address of an object, or its first non-static data member, is not defined (7.6.6 [expr.add]p4). For example, the following program has undefined behavior on the commented line:

```
#include <cstddef>
#include <cstdio>

struct T {
   int     i;
   double  j;
   short   k;
   void   *p;
};

int main() {
    using namespace std;

    T x = {};
    size_t y = offsetof(T, k);
    short *p = (short*)((byte*)&x + y);   _// `operator+` has undefined behavior_
    *p =123;
    printf("%d", x.k);
}
```

Note that the equivalent program in C has well-defined behavior, and every C++ compiler/library I have tried consistently produced the same behavior as the C program as their own manifestation of UB.

See [EMCS] Generalized PODs for a detailed discussion, and examples making well-defined use of the `offsetof` macro.

5 Proposed resolution

5.1 Adopt `offsetoff` as a keyword

Add `offsetof` to the list of keywords and non-overloadable operators. No *conforming* code breaks, as C++ users cannot define an `offsetof` macro, but can they define an `offsetof` function as long as they do not include any standard headers.

5.2 Specify semantics for the operator

Define the semantics of the `offsetof` operator under 7.6.2 [expr.unary]

Ensure errors are diagnosable, not UB.

5.3 Define pointer arithmetic with the result of `offsetof`

Define restricted pointer arithmetic on `std::byte*` pointing to the address of a standard layout class object, such that the increment would match a result from `offsetof` on that type. Similar arithmetic in non-standard layout classes is conditionally supported.

Note that, as far as we can tell, all current compilers already behave in this manner, even though the standard imposes no requirements on such a program.

5.4 Potential incompatibilities

As `offsetof` is a macro, it appears can be used in the predicate of a `#if` directive. However, to make sense of the *type* argument, the compiler must already have parsed source to the point that it knows that identifier is a class type, and that does not happen during preprocessing.

Users may have used `offsetof` as an identifier in their own code. However, valid use of this identifier by users is very restricted.

The implementation of the headers `<stddef.h>` and `<cstddef>`, when included by a C++ compiler, must avoid defining the `offsetof` macro if the C++ feature macro `cpp_offsetof_keyword` is defined.

6 Implementation experience

None.

7 Alternative resolutions

7.1 Make `<cstddef>` an importable header unit

This would allow us to `import std;` without requiring a `#include`, to allow users the bare minimum support to modernize their code, if they desire. The main benefit of `import` over `#include` though, other than supporting coding conventions, is to constrain the header to be idempotent, which would mostly affect the library implementers rather than the library consumers. Most of the other benefits of making this header an importable header unit were resolved by the introduction of the `std` library module, e.g., the definition of common type aliases such as `size_t` and `ptrdiff_t`.

7.2 Open a core issue

Open a core issue to make the pointer arithmetic well-defined.

7.3 Open a library issue

Open a LWG issue to diagnose functions and static data members, rather than UB.

8 Wording

All wording is relative to [N4944], the latest working draft at the time of writing.

N O T E : W O R D I N G T O B E D O N E

9 Acknowledgements

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

Thanks to Aaron Ballman for insights into the corresponding feature in C.

10 References

[EMCS] John Lakos, Vittorio Romeo, Rostislav Khlebnikov, and Alisdair Meredith. 2021. Embracing Modern C++ Safely.

[N4944] Thomas Köppe. 2023-03-22. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4944

[P2465R3] Stephan T. Lavavej, Gabriel Dos Reis, Bjarne Stroustrup, Jonathan Wakely. 2022-03-11. Standard Library Modules std and std.compat.
https://wg21.link/p2465r3

[P28654R0] Alisdair Meredith. 2023-05-15. Macros And Standard Library Modules.
https://wg21.link/p2654r0