`offsetof` Should Be a Keyword in C++26

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

Document #:	P2883R1
Date:	2025-01-10
Project:	Programming Language C++
Audience:	EWG
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

R1: January 2025 (pre-Hagenberg meeting)

— Removed the redundant section numbering in "Revision History"
— Applied intentional pagination
— Recorded review from EWG, Varna 2023
— Strengthened motivation following Varna review
  — Added discussion of [CWG2784]
— Discussed library solution using reflection, following St Louis meeting, 2024
— Added discussion of [P1839R6] and the follow-up [P3407R0]
— Added usage experience or its close approximation
— Provided complete wording

R0: Varna 2023

— Initial draft of this 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 Standard Library facilities that are specified as macros. Paper [P2654] 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 the `offsetof` feature is significant enough to merit investment of Committee time. In my opinion, integrating this macro into the Core language and resolving outstanding issues would significantly improve the C++ experience. The response to this paper, as a first paper on moving Standard Library macros into the language, can provide a good gauge of appetite for investing resources into such a project.

4 Stating the Problem

4.1 Macros mangle C++

Since specific macros are not direct language features but rather manipulate source text, surprising outcomes often occur when the C++ source code to be supplied as an argument to a macro includes the comma 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 braced initializers 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 solution does not work for the `offsetof` macro since neither argument supports parentheses.

Also, we need better clarity regarding whether a type expression, such as that produced by the `decltype` operator, is valid for the first argument or whether we must literally spell the name of a type as the token used preprocessing the macro. In practice, all the main front ends appear to accept the `decltype` form.

4.1.1 Simple example

In this example, we work through variations in the syntax of `offsetof` when trying to find the offset of the first data member of a standard-layout class template. The complexities are entirely due to `offsetof` being a macro, not a Core-language operator.

```
import std;
#include <cstddef>  // not an importable header unit so must #include

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~~
         // comma in the type name separates into multiple macro arguments

~~static_assert(offsetof((Test< int,int>), data) == 0); // error~~
         // parenthetical expression is not a type

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


static_assert(offsetof( decltype(Test<int, int>{}), data) == 0);  // OK?
       // avoids issues with commas in types used as macro arguments
       // needs a default-constructible type argument, which is true in this case
       // Can we use the decltype operator, or must the type be literal?

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

4.2 Macros cannot be exported

The macro `offsetof` is defined in the `<cstddef>` header, which is not an importable header unit. Although the contents of this header are exported from the Standard Library module `std`, such exports cannot include macros since 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. This restriction is explicitly acknowledged in the Standard by the following note (16.4.2.4 [std.modules]):

> [*Note 2:* Like all named modules, the C++ library modules do not make macros visible (10.3), such as `assert` (19.3.2), `errno` (19.4.2), `offsetof` (17.2.1), and `va_arg` (17.13.2). *—end note*]

If `<cstddef>` were an importable header unit, then macros would be available by importing that header unit (15.5 [cpp.import]), but since that is not the case in C++23, users must directly `#include` this header to access the `offsetof` macro.

Requiring the explicit use of `#include` to access features of the Core language does not play well with a long-term goal of moving C++ past the preprocessor as part of how we build code.

4.3 Obscure drafting

The current specification for the `offsetof` macro draws heavily on its specification in the C Standard, and that wording does not seem fully formed for the C++ Standard, notably in its use of the term *member-designator* that is never defined for C++. This ambiguity is at the heart of [CWG2784], which also raises the question of whether `offsetof` produces a core constant expression such that it might be used in `static_assert` statements?

4.4 Gratuitous undefined behavior

The valid set of arguments to the `offsetof` macro are restricted. It is conditionally supported for the first (type) argument to be a type other than a standard-layout class type; note that "conditionally supported" requires nonsupported cases to be a diagnosable error, so no UB should occur 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.

The interpretation of this argument is also a statically determined property at the point the macro is invoked and thus should be a diagnosable error in C++. Note that this situation cannot arise in C.

4.5 Result cannot be used in pointer arithmetic

EWG Update: Following its initial discussion of this paper at the Varna meeting in 2023, EWG recommended that this part of the paper be pursued independently via the already EWG-approved [P1839R6]. That paper, in turn, has split the specific concerns of `offsetof` into its own follow-up paper, [P3407R0].

The result of the `offsetof` macro cannot be used with regular pointer arithmetic to produce a pointer with the address of a nonstatic data member since pointer arithmetic on the address of an object or on its first nonstatic data member is not defined (see 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 and library I have tried consistently produced, as their own manifestation of UB, the same behavior as the C program.

See the “Generalized PODs” section of [EMCPPS] for a detailed discussion and for examples making well-defined use of the `offsetof` macro.

5 Proposed Resolution

Resolve [CWG2784] by fully adopting `offsetof` as a keyword and operator in C++.

5.1 Drafting plan

5.1.1 Adopt `offsetoff` as a keyword

Add `offsetof` to the list of keywords; it will be a nonoverloadable operator like `sizeof`, `alignof`, and `noexcept`. No *conforming* code is expected to break since C++ users cannot define an `offsetof` macro, but they can define an `offsetof` function as long as they do not include any standard headers.

5.1.2 Specify semantics for the operator

Define the semantics of the `offsetof` operator as a subclause of 7.6.2 [expr.unary]. We will explicitly support any valid *type_id* that represents a class type, in conjunction with an *identifier* that names a nonstatic data member of that class.

All errors will be diagnosable, not undefined behavior.

5.1.3 Add a feature macro

Add a new feature macro, `__cpp_offsetof_keyword`, to the list of predefined macro names.

5.1.4 Remove `offsetof` macro from relevant headers

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

This change is similar to the traditional treatment of Boolean types and literals in `<cstdbool>` and `<stdbool.h>`, of the C++ alternative keywords in `<iso646.h>`, of `static_assert` in `<cassert>` and `<assert.h>`, and so on.

5.1.5 Update all references that assume `offsetof` is a macro

There are multiple references to `offsetof` in the Standard that refer to its definition as a macro. Those references should either be updated to refer to the keyword or struck entirely.

5.2 Delegated features

The following parts of the original proposal are deferred to [P1839R6] at the request of EWG. Note that the author of that paper has further separated, into a stand-alone paper ([P3407R0]), the concerns specific to `offsetof` that are distinct from the more general topic of accessing bytes within an object's representation.

5.2.1 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. Since standard-layout classes cannot have virtual functions or virtual bases, implementations that choose to conditionally support such types are responsible for defining the result of `offsetof` in those cases.

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

5.3 Potential incompatibilities

5.3.1 Using the `offsetof` macro in preprocessing predicates

Since `offsetof` is a macro, it seems, at first, to be usable in the predicate of a `#if` directive. However, during phase 4 of translation, all identifiers that do not name macros when evaluated for the purposes of an `#if` predicate are treated as the literal `0` and do not satisfy the requirements of arguments to the `offsetof` macro.

5.3.2 Checking if `offsetof` is defined as a macro in preprocessing predicates

Valid code is permitted to test whether `offsetof` is defined as a macro by `#if defined(offsetof)`, indicating that either `<cstddef>` or `<stddef.h>` might have been transitively included by another header. I am not aware of any code deploying this trick.

5.3.3 User-supplied entities named `offsetof`

Users may have used `offsetof` as an identifier in their own code. However, given its nature as a Standard Library macro, users' valid employment of `offsetof` as an identifier is very restricted and is unlikely in practice.

6 Implementation Experience

6.1 As specified

We have no experience with the keyword spelled `offsetof`.

6.2 A keyword with a different spelling

Test programs have been built and run using the common intrinsic spelling of `__builtin_offsetof` *without* including any headers. This testing was run with Clang, GCC, and the EDG front end via the Godbolt Compiler Explorer. I do not know the equivalent spelling to test with the Microsoft compiler.

Using this built-in spelling, we verified that errors are diagnosed when the *identifier* argument names a static data members or a member-functions; these implementations had no undefined behavior.

Further testing ensured diagnostics for the named *identifier* being

— missing
— an enumerator
— an enumeration type
— an `enum class`
— a member class
— a type alias
— private data

When testing with the name of a reference member, only EDG issued an error. However Clang, GCC, and EDG all provided a warning that the class type was not a standard-layout class, although such a warning is not strictly required for conditionally supported constructs.

Most of these diagnostics reported that the member could not be found. A better QoI might be to report that the member was found to be an entity of the wrong kind.

Additional testing demonstrated that class templates with multiple arguments can be named without needing parentheses to protect the commas. Classes can be local classes, member classes, and unions. Incomplete classes are diagnosed.

In all cases, valid results were verified to be core constant expressions by using the result as an array bound and as the predicate to a `static_assert` declaration.

6.3 What remains to be done

To fully demonstrate usage experience, an update to one of the open-source compilers should

— add the feature macro
— replace the spelling of `_builtin_offsetof` with the simpler `offsetof`
— update the headers `<stddef.h>` and `<cstddef>` to avoid defining the `offsetof` macro unless the feature macro is not defined

Finally, the test compiler should build a nontrivial set of code, such as large open-source projects.

Note that this hypothetical implementation would still be no more than a proof of concept; a production-ready implementation would come with its own extensive test suite.

7 Alternative Resolutions

Note that none of these resolutions resolve any part of [CWG2784].

7.1 Provide a library-only solution through reflection

A library solution *is* coming through the `offset_of` meta function in the reflection proposal ([P2996R8]) that is tentatively on schedule for C++26, as long as it can complete the wording group reviews in time.

This function may be the recommended direction for C++ use of offset-related functionality but does not address the use of `offsetof` in existing code that would benefit from having a clear specification and the elimination of an unnecessary undefined behavior.

7.2 Make `<cstddef>` an importable header unit

This approach would allow us to employ `import std;` — without requiring an `#include` statement — to allow users the bare minimum support to modernize their code, if they desire. The main benefit of `import` over `#include`, however, 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`.

7.3 Open a Core issue for pointer arithmetic

Open a Core issue to make the pointer arithmetic well defined. See also [P1839R6] and [P3407R0].

7.4 Open a Standard Library issue to remove UB

Open a LWG issue to diagnose trying to find the offset of member-functions and static data members, rather than leaving as UB.

8 Reviews

8.1 EWG: 2023-06-16 Varna meeting

At the Varna meeting in 2023, EWG raised concerns regarding macro name clashes.

EWG offered clear direction that this paper should not attempt to solve the pointer arithmetic issues and should trust that those concerns will be separately resolved by [P1839R6], which has already passed EWG and is in Core review.

EWG also suggested that a Standard Library solution, with a distinct name, might be better. This approach is fundamentally untenable in C++03 since the C++ grammar does not support member names as parameters. However, those concerns are addressed by the reflection proposal for C++26, which indeed offers a more detailed `offset_of` function that provides not just the offset in bytes, but also the specific bit within a byte where that precision becomes relevant, such as in bit-fields.

EWG would like to see more motivation for solving such a minor feature with full language treatment.

The group expressed concern about lack of implementation experience with this identifier as a keyword, despite the common understanding that the macro is implemented with an intrinsic that models the proposed direction, i.e., is essentially the `offsetof` keyword with a different spelling that is reserved to the implementation.

The following poll was taken.

EWG encourages more work in the direction of P2883 (`offsetof` Should be a Keyword in C++26)

```
SF  F   N   A   SA
0   11  8   4   1
```

A summary of the Against votes suggested that consensus might be increased if a library solution is explored and if the motivation is clarified.

Note that a library solution is not possible using just C++23 syntax. The goal of being a drop-in replacement so that existing code would not need to change is lost, since the only way we know to pass a type argument is as a template parameter. However, the real problem is that the language offers no way to pass an *identifier* as a function argument; that part truly does need language support. Such language support is on its way in the form of reflection ([P2996R8]) but is neither a drop-in replacement nor an approved feature of C++26 at the time of writing.

9 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5001], the latest draft at the time of writing.

5.12 [lex.key] Keywords

Add the `offsetof` keyword to Table 5 — Keywords [tab:lex.key]

6.3 [basic.def.odr] One-definition rule

14 A definition of a class shall be reachable in every context in which the class is used in a way that requires the class type to be complete.

[*Example 6:* The following complete translation unit is well-formed, even though it never defines `X`:

```
struct X;               // declare X as a struct type
struct X* x1;           // use X in pointer formation
X* x2;                  // use X in pointer formation
```

—*end example*]

[*Note 3:* The rules for declarations and expressions describe in which contexts complete class types are required. A class type `T` must be complete if

(14.1) — an object of type `T` is defined (6.2), or
(14.2) — a non-static class data member of type `T` is declared (11.4), or
(14.3) — …
(14.11) — an lvalue of type `T` is assigned to (7.6.19), or
(14.12) — the type `T` is the subject of an `alignof` expression (7.6.2.6), or
(14.1x) — the type `T` is the subject of an `offsetof` expression (7.6.2.X), or
(14.13) — an *exception-declaration* has type `T`, reference to `T`, or pointer to `T` (14.4). —*end note*]

7.6.2 [expr.unary] Unary expressions

7.6.2.1 [expr.unary.general] General

1 Expressions with unary operators group right-to-left.

unary-expression:
    postfix-expression
    unary-operator cast-expression
    `++` *cast-expression*
    `--` *cast-expression*
    await-expression
    `sizeof` *unary-expression*
    `sizeof (` *type-id* `)`
    `sizeof ... (` *identifier* `)`
    `alignof (` *type-id* `)`
    `offsetof (` *type-id* `,` *identifier* `)`
    noexcept-expression
    new-expression
    delete-expression

unary-operator: one of
    `* & + - ! ~`

Review note: *The following wording is based on the form of* ***`alignof`*** *and the C specification for the* *`offsetof`* *macro.*

7.6.2.X [expr.offsetof] Offsetof

1 An `offsetof` expression yields the offset in bytes from the beginning of any object of type given by the *type-id* operand to a non-static data member of that type, nominated by the *identifier* operand. The *type-id* operand shall be a complete standard-layout class (11.2 [class.prop]). The *identifier* shall name an accessible non-static data member of that class.

2 The result is a prvalue of type `std::size_t`.

[*Note 1:* An `offsetof` expression is an integral constant expression (7.7 [expr.const]). The type `std::size_t` is defined in the standard header `<cstddef>` (17.2.1 [cstddef.syn], 17.2.4 [support.types.layout]). —*end note*]

3 [*Note 2:* The `offsetof` operator has the same semantics as the `offsetof` macro in the C standard library header `<stddef.h>` but accepts a restricted set of C++ type arguments in this document that are a superset of the types expressible in C. —*end note*]

9.3.2 [dcl.name] Type names

1 To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `offsetof,` `new`, or `typeid`, the name of a type shall be specified. This can be done with a *type-id* or *new-type-id* (7.6.2.8), which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

13.8.3.3 [temp.dep.expr] Type-dependent expressions

4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

literal
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
`offsetof (` *type-id* `,` *identifier* `)`
`typeid (` *expression* `)`
`typeid (` *type-id* `)`
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*
`throw` *assignment-expression*$_{opt}$
`noexcept (` *expression* `)`
requires-expression

~~[*Note 1:* For the standard library macro `offsetof`, see 17.2. —*end note*]~~

13.8.3.4 [temp.dep.constexpr] Value-dependent expressions

2 An *id-expression* is value-dependent if - (2.1) it is a concept-id and any of its arguments are dependent, - (2.2) it is type-dependent, - (2.3) it is the name of a non-type template parameter, - (2.4) it names a static data member that is a dependent member of the current instantiation and is not initialized in a *member-declarator*, - (2.5) it names a static member function that is a dependent member of the current instantiation, or - (2.6) it names a potentially-constant variable (7.7) that is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the *unary-expression* or expression is *type-dependent* or the *type-id* is dependent:

`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`typeid (` *expression* `)`
`typeid (` *type-id* `)`

```
alignof ( type-id )
offsetof ( type-id , identifier )
```

[*Note 1:* For the standard library macro offsetof, see 17.2. *—end note*]

15.11 [cpp.predefined] Predefined macro names

Add `__cpp_offsetof_keyword` to Table 22 — Feature-test macros [tab:cpp.predefined.ft].

16.4.2.3 [headers] Headers

6 Names which are defined as macros in C shall be defined as macros in the C++ standard library, even if C grants license for implementation as functions.

[*Note 2:* The names defined as macros in C include the following: `assert`, ~~`offsetof`,~~ `setjmp`, `va_arg`, `va_end`, and `va_start`. *—end note*]

16.4.2.4 [std.modules] Modules

6 *Recommended practice:* Implementations should avoid exporting any other declarations from the C++ library modules.

[*Note 2:* Like all named modules, the C++ library modules do not make macros visible (10.3), such as `assert` (19.3.2), `errno` (19.4.2), ~~`offsetof` (17.2.1),~~ and `va_arg` (17.13.2). *—end note*]

17.2 [support.types] Common definitions

17.2.1 [cstddef.syn] Header `<cstddef>` synopsis

```
// all freestanding
namespace std {

...

}

#define NULL see below
~~#define offsetof(P, D) see below~~
```

17.2.4 [support.types.layout] Sizes, alignments, and offsets

1 The macro `offsetof(type, member-designator)` has the same semantics as the corresponding macro in the C standard library header `<stddef.h>`, but accepts a restricted set of *`type`* arguments in this document. Use of the `offsetof` macro with a *`type`* other than a standard-layout class (11.2 [class.prop]) is conditionally-supported.[1] The expression `offsetof(type, member-designator)` is never type-dependent (13.8.3.3) and it is value-dependent (13.8.3.4) if and only if *`type`* is dependent. The result of applying the `offsetof` macro to a static data member or a function member is undefined. No operation invoked by the `offsetof` macro shall throw an exception and `noexcept(offsetof(type, member-designator))` shall be `true`.

2 The type `ptrdiff_t` is ...

C.8.3.X [diff.header.stddef.h] Header `<stddef.h>`

1 The token `offsetof` is a keyword in C++ (5.12) and is not introduced as a macro by `<stddef.h>` (17.14.4).

[1] ~~162) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.~~

C.8.5.2 [diff.offsetof] Macro `offsetof(`*type*`,` *member-designator*`)`

1 The macro `offsetof` defined in `<cstddef>` (17.2.1), accepts a restricted set of type arguments in C++. Subclause 17.2.4 describes the change.

10 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown; to Aaron Ballman for insights into the corresponding feature in C; and to Lori Hughes for reviewing this paper and providing editorial feedback.

11 References

[CWG2784] Corentin Jabot. 2023-08-21. Unclear definition of member-designator for offsetof.
https://wg21.link/cwg2784

[EMCPPS] John Lakos, Vittorio Romeo, Rostislav Khlebnikov, and Alisdair Meredith. 2021. *Embracing Modern C++ Safely*. Addison-Wesley.
https://emcpps.com

[N5001] Thomas Köppe. 2024-12-17. Working Draft, Programming Languages — C++.
https://wg21.link/n5001

[P1839R6] Brian Bi, Krystian Stasiowski, Timur Doumler. 2024-10-14. Accessing object representations.
https://wg21.link/p1839r6

[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

[P2654] Alisdair Meredith. Macros And Standard Library Modules.
https://wg21.link/p2654

[P2996R8] Barry Revzin, Wyatt Childers, Peter Dimov, Andrew Sutton, Faisal Vali, Daveed Vandevoorde, Dan Katz. 2024-12-17. Reflection for C++26.
https://wg21.link/p2996r8

[P3407R0] Brian Bi. 2024-10-14. Make idiomatic usage of 'offsetof' well-defined.
https://wg21.link/p3407r0