std::containing_object_of_member

Document #:	TODO
Date:	2021-11-12
Project:	Programming Language C++
Audience:	Library Working Group

Reply-to:	Michael Scire <sciresm@gmail.com>

Contents

1 Abstract

This paper proposes the library utility function `std::containing_object_of_member`, which would allow one to use a pointer-to-non-static-data-member and a reference to a member subobject to obtain a reference to the object which contains the member subobject. This utility has much value (particularly in code working with intrusive member objects), and cannot currently be implemented without undefined behavior. Existing code to implement a form of it using `offsetof` became undefined behavior in C++17.

2 Introduction / Motivation

When working with intrusive member subobjects, one of the fundamental required operations is to obtain from a reference to the intrusive member subobject a reference to the object which contains the intrusive member subobject.

For example, we can imagine the following as a way to implement generic lists of objects:

```
struct intrusive_list_node {
    intrusive_list_node *prev;
    intrusive_list_node *next;

    // ...
};

class intrusive_list_impl {
    // Common implementation of iterator/list operations on intrusive_list_nodes
    // ...
};

template<auto>
class intrusive_list;
```

```
template<typename S, intrusive_list_node S::*MemberPtrToNode>
class intrusive_list<MemberPtrToNode> {
    // Thin wrapper around intrusive_list_impl for a specific intrusive_list_node member.
    // ...
};

struct example {
    intrusive_list_node list_node_for_usecase_x;
    intrusive_list_node list_node_for_usecase_y;
    int data;
};

// The example object may simultaneously be in two lists which operate
// on its respective list node members.
using example_list_for_usecase_x = intrusive_list<&example::list_node_for_usecase_x>;
using example_list_for_usecase_y = intrusive_list<&example::list_node_for_usecase_y>;
```

In order to implement the class `intrusive_list<MemberPtrToNode>`, it is necessary to have some way of obtaining the object which contains an `intrusive_list_node` member from a reference to the node and a member pointer.

That is to say, the following primitive is needed:

```
template<typename S>
S &containing_object_of_node(intrusive_list_node S::*pmd, intrusive_list_node &node);
```

Unfortunately, there is no way to implement this primitive in user code, as the "obvious" implementation invokes undefined behavior:

```
template<typename S>
S &containing_object_of_node(intrusive_list_node S::*pmd, intrusive_list_node &node) {
    // Get the address of the node
    const uintptr_t node_address = reinterpret_cast<uintptr_t>(&node);

    // Get and return the address of the containing object
    // This is UB!
    const uintptr_t ofs = reinterpret_cast<uintptr_t>(&(reinterpret_cast<S *>(0)->*pmd));
    const uintptr_t containing_object_address = node_address - ofs;
    return *reinterpret_cast<S *>(containing_object_address);
}
```

Indeed, even if one is willing to set aside templates and attempt to use the `offsetof` macro, the function cannot be implemented in user code since C++17:

```
example &containing_object_of_node_for_usecase_x(intrusive_list_node &node) {
    // Get the address of the node
    const uintptr_t node_address = reinterpret_cast<uintptr_t>(&node);

    // Get and return the address of the containing object
    // This is UB since C++17
    const auto ofs = offsetof(example, list_node_for_usecase_x);
    const uintptr_t containing_object_address = node_address - ofs;
    return *reinterpret_cast<example *>(containing_object_address);
}
```

This paper proposes to remedy this (and allow fixing the previously-non-UB offsetof logic that much existing code

uses) by adding a new function (declared in the `<type_traits>` header) named `std::containing_object_of_member` that takes a pointer to member data and a member reference and returns a reference to the object which contains the member.

3 Design considerations

This paper proposes that given any pointer to member data and a reference to the type which said pmd points to, `std::containing_object_of_member` would return a reference to the type of the container for the pmd.

The two primary design questions that seem to arise are whether to support pointer to member data for non Standard Layout class, and whether or not `std::containing_object_of_member` should be constexpr.

This paper proposes to support both; support for pointer-to-member data for non Standard Layout maximizes the usefulness of the function and seems okay since the layout details are not exposed to the caller in any way.

Similarly, with constexpr new supported, it can be desirable to work with lists (and similar data structures) at compile-time.

In order to support changing the active member of a union during constant evaluation, the standard already implicitly requires compilers to be able to detect whether an arbitrary access is one to a member subobject of some containing object; given this and that providing such a utility would enable most code using intrusive members to "just work", it seems desirable to support.

A constexpr implementation would require compiler magic to support; however, compiler magic is already required in order to implement this function, and so this seems acceptable.

The paper further notes that the function is difficult to name, and `std::containing_object_of_member` is slightly wordy, but that the proposed name mirrors well-known uses of the functionality it provides:

— The Linux kernel macro equivalent to the proposed functionality is `container_of`.
— The Windows macro equivalent to the proposed functionality is `CONTAINING_RECORD`.

4 Proposed Wording

In Header synopsis 20.15.3 [meta.type.synop], add:

// [meta.member], member relationships

```
template<class T, class U>
constexpr T &containing_object_of_member(U T::*pmd,
    U &member_subobject) noexcept;

template<class T, class U>
constexpr T const &containing_object_of_member(U T::*pmd,
    U const &member_subobject) noexcept;

template<class T, class U>
constexpr T volatile &containing_object_of_member(U T::*pmd,
    U volatile &member_subobject) noexcept;

template<class T, class U>
constexpr T const volatile &containing_object_of_member(U T::*pmd,
    U const volatile &member_subobject) noexcept;
```

Constraints: `containing_object_of_member` should participate in overload resolution only if pmd is a pointer to a non-static data member (that is, if `std::is_member_object_pointer_v<decltype(pmd)>` is true).

Returns: If `member_subobject` is a member subobject (11.4 [class.mem]) of some object `o` such that `o->*pmd` refers to the same object as `member_subobject`, a reference to `o` with the appropriate cv qualification corresponding to `member_subobject`. Otherwise, undefined behavior.