# Member Templates for Local Classes

Document Number: P2044R2
Date: 2020-04-05
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: Evolution

## Abstract

This paper proposes that local classes be allowed to have member templates.

## Tony Tables

| Before | After |
|---|---|
| `namespace detail {`<br>`template<typename Receiver>`<br>`struct error_to_exception_receiver {`<br><br><br>`[[noreturn]]`<br>`  void set_done() {`<br>`    throw std::system_error(`<br>`      make_error_code(`<br>`        std::errc::operation_canceled));`<br>`  }`<br>`  [[noreturn]]`<br>`  void set_error(std::error_code ec) {`<br>`    throw std::system_error(ec);`<br>`  }`<br>`  template<typename... Args>`<br>`  void set_value(Args&&... args)`<br>`    noexcept(/* ... */)`<br>`  {`<br>`    inner.set_value(std::forward<Args>(`<br>`      args)...);`<br>`  }`<br>`  Receiver inner;`<br>`};`<br>`}`<br>`template<typename Receiver>`<br>`auto error_to_exception(Receiver r)`<br>`  noexcept(/* ... */)`<br>`{`<br>`  return detail::error_to_exception_receiver<`<br>`    Receiver>{std::move(r)};`<br>`}` | `template<typename Receiver>`<br>`auto error_to_exception(Receiver r)`<br>`  noexcept(/* ... */)`<br>`{`<br>`  struct receiver {`<br>`    [[noreturn]]`<br>`    void set_done() {`<br>`      throw std::system_error(`<br>`        make_error_code(`<br>`          std::errc::operation_canceled));`<br>`    }`<br>`    [[noreturn]]`<br>`    void set_error(std::error_code ec) {`<br>`      throw std::system_error(ec);`<br>`    }`<br>`    template<typename... Args>`<br>`    void set_value(Args&&... args)`<br>`      noexcept(/* ... */)`<br>`    {`<br>`      inner.set_value(std::forward<Args>(`<br>`        args)...);`<br>`    }`<br>`    Receiver inner;`<br>`  };`<br><br><br><br><br><br>`  return receiver{std::move(r)};`<br><br>`}` |
| `template<typename... Args>`<br>`void output_variant(std::ostream& os,`<br>`  const std::variant<Args...>& v)`<br>`{`<br>`  auto vis = [&os](const auto& obj) noexcept(` | `template<typename... Args>`<br>`void output_variant(std::ostream& os,`<br>`  const std::variant<Args...>& v)`<br>`{`<br>`  struct {` |

```
    std::is_same_v<std::decay_t<decltype(obj)>,
    std::monostate>)
  {
    if constexpr (!std::is_same_v<std::decay_t<
      decltype(obj)>, std::monostate>)
    {
      os << obj;
    }
  };
  std::visit(vis, v);
}
```

```
    std::ostream& os;
    void operator()(const std::monostate&) const
      noexcept {}
    template<typename T>
    void operator()(const T& obj) const
    {
      os << obj;
    }
  } vis{os};
  std::visit(vis, v);
}
```

## Motivation

It is good programming practice to limit or eliminate visibility of implementation details. For this purpose C++ has the concept of internal linkage, no linkage, and member accessibility. Unfortunately the restriction that local classes not have member templates limits the applicability of some of these.

Binding names to the most enclosing scope in which there is a need to reference them eliminates name collisions. For this reason C++ has the concept of namespaces along with separate name lookup scopes within classes and functions.

Local classes provide a means to leverage both of these: A local class has no linkage and is invisible to all code outside its containing function. A local class is scoped to its enclosing function and cannot be ambiguated by other non-local declarations.

Unfortunately C++ limits local classes by preventing them from having member templates. Therefore classes that are logically an implementation detail of a function but also have a member template must be placed at class or namespace scope. If the class is placed at namespace scope one must make a context dependent decision: In a source file one should place it in an anonymous namespace (to give it internal linkage). In a header file one should place the class in a namespace which contains implementation details by convention (e.g. namespace detail).

These techniques are limiting: Internal linkage can lead to name conflicts when a “unity” build is performed. Employing a namespace which contains implementation details by convention does not actually prevent consumers from using and relying on its contents (i.e. Hyrum’s Law).

## Background

In C++98 local classes could not be passed as template parameters. This restriction was lifted in C++11 [1] as was necessary for lambdas to be useful [2]. At this time consideration was given to allowing local classes not only to have member templates but also to be themselves templates and to allow specializations thereof. All three of these possibilities were dismissed as "any of these would require a syntax change" [1].

C++14 added generic lambdas. A lambda expression that is a generic lambda is an expression of closure type where the closure type is a local class with a member template. An early paper on generic lambdas commented that "we'd need to ensure that the semantics of member templates of local classes are well defined and consistent with those of member templates of non-local classes [...] before this feature can be incorporated" [3]. A later paper drops this seemingly without comment and adds a narrow exception to §13.6.2 [temp.mem] permitting closure types to have member templates but no other local classes [4].

## Syntax Change

Allowing local classes to have member templates does not require a syntax change. The language provides that:

- *function-body* may contain a *compound-statement* (§9.4.1 [dcl.fct.def.general])
- *compound-statement* may contain one or more *statement* (by way of *statement-seq*) (§8.3 [stmt.block])
- *statement* can be a *declaration-statement* (clause 8 [stmt.stmt])
- *declaration-statement* is a *block-declaration* (§8.7 [stmt.dcl])
- *block-declaration* can be a *simple-declaration* (clause 9 [dcl.dcl])
- *simple-declaration* can contain a *decl-specifier* (by way of *decl-specifier-seq*) (clause 9 [dcl.dcl] & §9.1 [dcl.spec])
- *decl-specifier* can be a *defining-type-specifier* (§9.1 [dcl.spec])
- *defining-type-specifier* can be a *class-specifier* (§9.1.7 [dcl.type])
- *class-specifier* can contain a *member-specification* (clause 10 [class])
- *member-specification* can contain a *member-declaration* (§10.3 [class.mem])
- *member-declaration* can be a *template-declaration* (§10.3 [class.mem])

We can also arrive at this conclusion without walking the language's grammar: If the syntax did not allow for member templates within local classes it would be unnecessary for §13.6.2 [temp.mem] to disallow them as they could not occur in the first place.

## Modules & Reachability

A contemporary paper of the first revision of this paper [5] raised the issue of the reachability of member templates of local classes. §10.4 [module.global.frag] specifies that member templates of local classes would be reachable from the caller of the function producing the local class:

"*A declaration D is decl-reachable from a declaration S in the same translation unit if [...] D does not declare a function or function template and S contains [a] [...] type-name [...] naming D [...]*"

In reading the above we imagine that D is the declaration of the local class (not the member template) and S is the declaration of the function which produces it.

"*A declaration D is decl-reachable from a declaration S in the same translation unit if [...] there exists a declaration M that is not a namespace-definition for which M is decl-reachable from S and [...] one of M and D declares a class [...] C and the other declares a member [...] of C [...]*"

In reading the above we imagine that D is the declaration of some member template, M is the declaration of the local class, and C is the local class.

Note that we can less rigorously arrive at this conclusion by analogy with generic lambdas: Their function call operators (which are member templates) are reachable from contexts outside the function in which they originate. Also by analogy with non-template members of local classes: They are reachable from contexts outside the function in which they originate.

## Implementability

The author spoke with implementers from the following compilers:

- Clang
- MSVC

The only issue raised was that the eager substitution behavior of Clang may surprise some users.

## Eager Substitution

The following code currently does not compile on Clang (whereas it is accepted by GCC and MSVC):

```
template<typename T>
auto func(T t) {
    return [](auto) { decltype(t) u = 123; };
}
int main() {
    auto f = func(nullptr);
}
```

Note that this does not compile despite the fact that we never call the returned lambda (and thus never instantiate the member template of the generic lambda). Calling the returned lambda causes all three compilers to reject this code.

It has been suggested that this behavior of eagerly substituting through the body of local classes with member templates could surprise users especially in the context of code guarded by if constexpr or constrains clauses.

During EWG-I review it was indicated that this behavior is acceptable as the code above is “ill-formed, no diagnostic required.” The following excerpt from §13.8 [temp.res] is relevant:

“*The validity of a template may be checked prior to any instantiation. The program is ill-formed, no diagnostic required, if [...] no valid specialization can be generated for a template or a substatement of a constexpr if statement within a template and the template is not instantiated, or [...] no substitution of template arguments into a type-constraint or requires-clause would result in a valid expression, or [...] a hypothetical instantiation of a template immediately following its definition would be ill-formed due to a construct that does not depend on a template parameter [...]*”

## Feature Test Macro

This paper proposes a feature test macro: `__cpp_local_class_member_templates`. During review in EWG-I motivation for this feature test macro was requested. It was unclear to the room what code would actually make use of the feature test macro. The thinking was that if one wasn’t sure whether or not local class member templates were supported one would just directly write a class in an anonymous or detail namespace.

A motivating example for the feature test macro is a situation where a “sink” function is an acceptable fallback implementation but forwarding may be more ideal. Consider the following example (which admittedly could be a generic lambda due to its triviality):

```
template<typename T>
auto make_back_insert_function(T& container) noexcept {
    struct inserter {
        T& container;
#if defined(__cpp_local_class_member_templates) && \
  (__cpp_local_class_member_templates >= some value)
        template<typename U>
        void operator()(U&& u) const {
            container.emplace_back(std::forward<U>(u));
#else
        void operator()(typename T::value_type e) const {
            container.push_back(std::move(e));
#endif
        }
    };
    return inserter{container};
}
```

This motivation can be understood by analogy with uses of the following macros from Boost which make move or forwarding semantics available in the presence of C++11 but fall back to copy semantics otherwise:

- `BOOST_ASIO_MOVE_ARG`
- `BOOST_ASIO_MOVE_CAST`
- `BOOST_NO_CXX11_RVALUE_REFERENCES`

## Proposed Wording

§13.6.2/2 [temp.mem]:

*~~A local class of non-closure type shall not have member templates.~~ Access control rules apply to member template names. A destructor shall not be [...]*

Add to the table in §14.8 [cpp.predefined]:

| ***Macro name*** | ***Value*** |
|---|---|
| *`__cpp_local_class_member_templates`* | *some value* |

## Acknowledgements

The author would like to thank Brian Rivas, Nathan Myers, Thomas Köppe, Richard Smith, and Gabriel Dos Reis for assistance in the preparation of and research for this paper.

## Revision History

### Revision 1

- Merged with P1988R0 as per EWG-I in Prague
    - Added section on modules & reachability
- Replaced section on implementations with a section on implementability
- Added section on eager substitution
- Added section on feature test macro

### Revision 2

- Updated Tony Tables (color & alignment for clarity)

## Review History

### Prague 2020

Revision 0 was presented to EWG-I. The following poll was taken:

*Merge with P1988, talk to other implementers, take the revised paper to EWG if no additional problems are discovered.*

```
SF F N A SA
2  8 2 0 0
```

## References

[1] A. Williams. Making Local Classes more Useful SC22/WG21/N1427=03-0009
[2] J. Willcock, J. Jarvi, D. Gregor, B. Stroustrup, and A. Lumsdaine. Lambda expressions and closures for C++ N1968=06-0038
[3] F. Vali, H. Sutter, and D. Abrahams. Proposal for Generic (Polymorphic) Lambda Expressions N3418=12-0108
[4] F. Vali, H. Sutter, and D. Abrahams. Proposal for Generic (Polymorphic) Lambda Expressions (Revision 2) N3559
[5] S. Downey. Allow Templates in Local Classes P1988R0