Member Templates for Local Classes

Document Number: P2044R0
Date: 2020-01-12
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: EWGI, Evolution

Abstract

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

Tony Tables

Before	After

namespace detail {
template<typename Receiver>
struct error_to_exception_receiver {
  [[noreturn]]
  void set_done() {
  throw std::system_error(
  make_error_code(
  std::errc::operation_canceled));
  }
  [[noreturn]]
  void set_error(std::error_code ec) {
  throw std::system_error(ec);
  }
  template<typename... Args>
  void set_value(Args&&... args)
  noexcept(/* ... */)
  {
  inner.set_value(std::forward<Args>(args)...);
  }
  Receiver inner;
};
}
template<typename Receiver>
auto error_to_exception(Receiver r)
  noexcept(/* ... */)
{
  return detail::error_to_exception_receiver<
  Receiver>{std::move(r)};
}
	template<typename Receiver>
	auto error_to_exception(Receiver r)
	  noexcept(/* ... */)
	{
	  struct receiver {
	  explicit receiver(Receiver inner)
	  noexcept(/* ... */)
	  : inner_(std::move(inner))
	  {}
	  [[noreturn]]
	  void set_done() {
	  throw std::system_error(
	  make_error_code(
	  std::errc::operation_canceled));
	  }
	  [[noreturn]]
	  void set_error(std::error_code ec) {
	  throw std::system_error(ec);
	  }
	  template<typename... Args>
	  void set_value(Args&&... args)
	  noexcept(/* ... */)
	  {
	  inner.set_value(std::forward<Args>(
	  args)...);
	  }
	  Receiver inner;
	  };
	  return receiver{std::move(r)};
	}

template<typename... Args>
void output_variant(std::ostream& os,
  const std::variant<Args...>& v)
{
  std::visit([&os](const auto& obj) noexcept(
  std::is_same_v<std::decay_t<decltype(obj)>,
  std::monostate>)
	template<typename... Args>
	void output_variant(std::ostream& os,
	  const std::variant<Args...>& v)
	{
	  struct visitor {
	  std::ostream& os;
	  template<typename T>

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

```
    void operator()(const T& t) const { os << t; }
    void operator()(const std::monostate&) const
      noexcept
    {}
  };
  visitor 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.

Implementations

An early paper on generic lambdas claims that Clang 3.2 supports local classes with member templates [3].

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 and Nathan Myers for assistance in the preparation of this paper.

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