Overload sets as function arguments

Andrew Sutton <asutton@uakron.edu>

Date: 2016-05-28

Document number: P0119R2

Revision history

- R0 – Initial draft
- R1 – Added new identifiers for operators. Forward arguments to calls. Add support for qualified lookup and class members.
- R2 – Reorganized. Added concepts to introduction. All return types explicitly specified.

Introduction

I want to be able to use overloaded names as arguments to generic algorithms. That is, I want this program to be well-formed.

```
template<typename T>
T twice(T x) { return x * x; }


template<typename I>
void f(I first, I last)
{
  transform(first, last, twice); // error: twice is not a function
}
```

Today, the program is ill-formed because no type can be deduced for that function argument. This is because `twice` is an overloaded name (a function template defines a family of functions), so there's no concrete type that can be assigned to that identifier.

Notice that if `twice` names a (single) function, then this just works. That's unfortunate because it adds an asymmetry to the language. In this use, names of functions work one way when not overloaded and another when they are. Note that this isn't the case for function calls. it doesn't matter if we name one function, a set of functions, or a function template. The syntax for using them is identical.

The inability to simply write `twice` also makes it harder for me to express my intent. The best I can do today is to use a lambda expression that calls the overload name.

```
template<typename I>
void f(I first, I last)
{
  transform(first, last, [](auto const& x) { return twice(x); });
}
```

This is OK. It works. But I would very much prefer to write just `twice`. Otherwise, I just bury the use of the function in boilerplate lambda code.

I could also have explicitly instantiated the function, but that only works for templates. If `twice` were a finite set of functions (like `abs`), then that would not work.

There's another very good reason to allow this: concepts. With concepts, we get the ability to define abbreviated function templates:

```
void is_odd(Integer n) // Integer is a concept
{
  return n % 2;
}
```

The function `is_odd` is an abbreviated function template. It is equivalent to having written this:

```
template<Integer T>
void is_odd(T n)
{
  return n % 2;
}
```

With the introduction of abbreviated function templates in concepts, it becomes more difficult to distinguish between non-overload and overloaded names. In other words, to pass `is_odd` as a predicate to say, `find_if`, you would have to wrap it in a lambda or instantiate it. Developers should not be required to know about every declaration (or absence of declarations) just to use a function. State what you want, and let the compiler find what you need.

Proposal

This paper proposes to allow passing overloaded names as arguments to generic algorithms. The reason for doing so is to give equal treatment, in terms of a common use pattern, to names that refer to individual functions, function templates, and sets of thereof.

The solution is straightforward. To support this feature, we extend template argument deduction to handle the case where an argument is an *id-expression* that names a set of functions, and the parameter's type is a type parameter. In this case, we synthesize a *lambda-expression* that calls the *id-expression* with a forwarded set of arguments.

```
T twice(Number n) // Abbreviated function template
{
  return n * n;
}

template<typename I>
```

```
void apply(I first, I last)
{
  std::transform(first, last, first, twice);
}
```

When resolving the call to `transform`, we can automatically transform the *unqualified-id* `twice` into the following generic generic lambda.

```
[](auto&&... args)
  -> decltype(twice(std::forward<decltype(args)>(args)...))
{
  return twice(std::forward<decltype(args)>(args)...);
};
```

The lambda takes a pack of forwarded arguments, allowing whatever values necessary. The lambda effectively defines a call to `twice` that forwards those arguments. The return type is written explicitly in order to provide better error checking.

Note that `twice` is still an *unqualified-id* in the lambda. This means that argument dependent lookup will apply. This makes sense because the user clearly wrote `f` as *unqualified-id* at the call site. The use of *qualified-id*s is discussed below.

The proposed mechanism does not affect any existing overload resolutions because it is only engaged to handle a new case. That is, it makes ill-formed programs well-formed. Deductions and conversions involving *id-expression*s that name a single function are *not* affected by this new feature.

Defining the feature by extending template argument deduction has another benefit: we can use this technique to create function object variables:

```
auto fn = twice;    // Declares a function object
std::cout << fn(3); // prints 6
```

The deduced type of `fn` will be the lambda closure type of the lambda described above. Note that if `twice` is not an overload, then the deduced type will be a pointer (or reference?) to that function.

Qualified lookup

Using a *qualified-id* that names an overload set results in qualified lookup.

```
void sort(first, last, N::f);
```

Yields this lambda:

```
[](auto&&... args)
  -> decltype(N::f(std::forward<decltype(args)>(args)...))
{
  return N::f(std::forward<decltype(args)>(args)...);
}
```

Member access

It would be nice if this worked for member functions too.

```
struct S
{
  void f(int&);
  void f(std::string&);
};

S s;
std::transform(first, last, s.f);
```

When synthesizing a lambda for a class member access, we need to capture only the complete object (`s`) and build the function call using the same *postfix-expression*.

```
[&s](auto&&... args)
  -> decltype(s.f(std::forward<Args>(args)...))
{
  return s.f(std::forward<Args>(args)...);
}
```

Operators

If we support normal function names, we should also support *operator-id*s. For example, writing this:

```
void sort(first, last, operator>);
```

would yield this lambda:

```
[](auto&& a, auto&& b)
  -> decltype(operator>(std::forward<decltype(a)>(a),
                        std::forward<decltype(b)>(b))
{
  return operator>(std::forward<decltype(a)>(a),
                   std::forward<decltype(b)>(b));
};
```

The lambda is not variadic because operator> must take exactly two arguments. The returned expression is defined in terms of the name `operator>`. Note that this *will not* find built-in relational operators. This is just how the language works.

As an alternative, I propose that we introduce a new set of identifiers that can be used in this context. For example, I would prefer to call `sort` like this:

```
std::sort(first, last, (>));
```

Here, the tokens `(>)` form an *operator-expression-id*. When deducing against an *operator-expression-id*, the lambda's return value is defined in terms of the named expression, and not the operator. That is, the lambda for the use above would be:

```
[](auto&& a, auto&& b)
  -> decltype(std::forward<decltype(a)>(a) > std::forward<decltype(b)>(b))
{
  return std::forward<decltype(a)>(a) > std::forward<decltype(b)>(b);
};
```

The arguments are still forwarded, because the normal resolution rules could find user-defined overloads that accept rvalue references.

Note that this feature also allows this usage:

```
auto gt = (>);
```

This proposal does not allow *operator-expression-id*s to be used to be used as the name of variables, functions, or parameters. They can only appear within an *initializer*.

For both *operator-function-id*s and *operator-expression-id*s, the rules for synthesizing lambdas depend on the *operator*. For those that have only a binary or unary form, we can synthesize the lambda directly, as we did above. For operators that have both unary and binary versions, we would need to synthesize a new lambda closure type that accepted either. That type might look like this:

```
struct polymprhic_lambda
{
  template<typename T>
  auto operator()(T&& x) const
    -> decltype(std::forward<T>(x))
  {
    return op std::forward<T>(x);
  }

  template<typename T, typename U>
  auto operator()(T&& a, U&& b) const
    -> decltype(std::forward<T>(a) op std:forward<U>(b));
  {
    return std::forward<T>(a) op std:forward<U>(b);
  }
}
```

Here `op` stands for the unary/binary operator.

The lambda used for the function call operator `(())` is this:

```
[](auto&& f, auto&&... arg)
  -> decltype(std::forward<decltype(f)>(f)(std::forward<decltype(arg)>(args)...))
{
  return std::forward<decltype(f)>(f)(std::forward<decltype(arg)>(args)...);
}
```

And for the subscript operator (`[]`), the lambda is this:

```
[](auto&& x, auto&& y)
  -> decltype(std::forward<decltype(x)>(x)[std::forward<decltype(y)>(y))
{
  return std::forward<decltype(x)>(x)[std::forward<decltype(y)>(y)];
}
```

I propose that `(++)` and `operator++` refer only to the pre-increment operators in these cases. Similarly for `(--)` and `operator--`. I am open to suggestion on how we might support the post-increment/decrement forms. One such suggestion is to simply not support those operations.

I think that allowing `new` and `delete` operators might be unwise. They are currently excluded from the proposal.

Proposed wording

5.1.1 General [expr.prim.general]

Add a new kind of *unqualified-id* named *operator-expression-id*.

```
unqualified-id:
  identifier
  operator-function-id
  operator-expression-id
  ...

operator-expression-id:
  ( operator )
```

An *operator-expression-id* denotes a set of expressions that can be used as a function argument as described in [temp.deduct.call]. They shall not appear in any context other than *initializer*s.

14.8.2.1 Deducing template arguments from a function call [temp.deduct.call]

Note: We want to synthesize a lambda expression from an *id-expression* in a very narrow set of cases. In particular, we must be performing deduction of an *id-expression* that names an overload set against an unadorned type template parameter or placeholder type (i.e., a plain

`T`) and not, for example, a type of the form `R(*)(Args...)`. Otherwise, these rules would conflict with paragraph 6. Add the following after paragraphs at the end of this section.

TODO: Finish writing the rules for *operator-function-id*s.

If `P` has type `T` where `T` is a type template parameter, and `A` is an *id-expression* that names a set of overloaded functions, deduction is performed against the expression defined by the following rules.

- If `A` is an *identifier* `f`, that expression is the *lambda-expression*:

```
[](auto&&... args)
  -> decltype(f(static_cast<decltype(args)>(args)...))
{
  return f(static_cast<decltype(args)>(args)...);
}
```

- If `A` is the *qualified-id* `N::f`, that expression is the *lambda-expression*:

```
[](auto&&... args)
  -> decltype(N::f(static_cast<decltype(args)>(args)...))
{
  return N::f(static_cast<decltype(args)>(args)...);
}
```

- If A is the *operator-expression-id* `(())`, the *lambda-expression* is

```
[](auto&& f, auto...&& args) -> decltype(auto)
{
  return static_cast<decltype(a)>(a)(static_cast<decltype(args)>(args)...);
}
```

- If A is the *operator-expression-id* is `([])`, that *lambda-expression* is

```
[](auto&& a, auto&& b) -> decltype(auto)
{
  return static_cast<decltype(a)>(a)[static_cast<decltype(b)>(b)];
}
```

- If A is an *operator-expression-id* whose *operator* is one of `+`, `-`, `*`, or `&`, that deduction is performed against a prvalue object of unique, unnamed, non-union class type that is equivalent to

```
struct closure_type
{
  template<typename T>
  auto operator()(T&& x) const
    -> decltype(@ static_cast<T>(x))
  {
    return @ static_cast<T>(x);
  }
```

```
  template<typename T, typename U>
  auto operator()(T&& a, U&& b) const
    -> decltype(static_cast<T>(a) @ static_cast<U>(b))
  {
    return static_cast<T>(a) @ static_cast<U>(b);
  }
}
```

where `@` is replaced by the *operator*.

- Otherwise, if A is any other *operator-expression-id*, that *lambda-expression* is

```
[](auto&& a, auto&& b)
  -> decltype(static_cast<decltype(a)>(a) @ static_cast<decltype(b)>(b))
{
  return static_cast<decltype(a)>(a) @ static_cast<decltype(b)>(b);
}
```

- Otherwise, the program is ill-formed.

Issues

- The proposal is missing synthesis rules for pre/post-increment and decrement.
- The current proposal does not support for *conversion-id*s or

Implementation experience

I started an implementation of this feature in GCC last year, but didn't finish it. Effectively, the implementation is capable of recognizing when to synthesize the lambda expression from an *id-expression*, but not actually synthesizing the lambda expression.

Related work

N3617 describes "lifting expressions", which satisfy many of the same aims of this proposal. However, it requires the *lambda-introducer* before the *id-expression*. This extra annotation seems unnecessary to me.

N3617 goes further and suggests that we allow projection functions like this:

```
struct user
{
  int id;
  std::string name;
};

vector<user> users{ {4, "A"}, {1, "B"}, {3, "C"}, {0, "D"}, {2, "E"} };
sort(users.begin(), users.end(), ordered_by([]id));
```

I think that the current trend is that this problem be solved in the library and not in the language. For example, the `sort` function could be extended to allow the following:

```
sort(users.begin(), users.end(), &user::id);
```

This would have the same effect as example given above, although it's not clear what `ordered_by` should actually do or how `id` resolves to the member variable.

N3701 made brief mention of this feature, more or less in the form that it is presented here. This paper incorporates the rules from N3617 for forming lambda expressions from operators.

Acknowledgments

Thanks to Florian Weber for comments on early drafts of this paper. Thanks to Tomasz Kamiński for his suggestions about return type deduction, class member access, and editorial support.