Overload sets as function arguments

Andrew Sutton <asutton@uakron.edu>

Date: 2016-02-12

Document number: P0119R1

Introduction

I want to be able to use the names of overloaded functions as arguments to algorithms. Suppose I have a generic algorithm that `transform`s a sequence of values by some function `f`. I want to write it like this:

```
template<typename I>
void apply_f(I first, I last)
{
  std::transform(first, last, first, f);
}
```

If `f` names a (single) function, then this *just works*, but if it names an overloaded set or a function template, the program is ill-formed. That's unfortunate. First, it's inconsistent: if `f` looks like a function, then I should be able to use it as a function. Second, there is no more obvious way to express my intent. The best I can do today is to use a lambda expression that then calls the overload set. Like this:

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

I would very much prefer to write just `f`.

This paper proposes the use of overload sets as function arguments and variable initializers. In addition to the use of functions above, we would also like to use operator names as well. For example, I want to call `sort` like this:

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

And this would be similar to writing:

```
std::sort(first, last, [](auto const& a, auto const& b) { return a > b; };
```

And I should be able to define function objects using the same technique:

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

This feature can be provided without introducing runtime overhead.

How it works

The mechanism that makes this language feature work is to extend template argument deduction to handle cases where a function argument is an *id-expression* naming an overload set, and to synthesize a *lambda-expression* for the *id-expression*.

In this example:

```
template<typename T>
void f(T&);

template<typename I>
void apply_f(I first, I last)
{
  std::transform(first, last, first, f);
}
```

The *unqualified-id* `f` refers to a set of overloaded functions (a template is a family of functions). We can automatically transform *unqualified-id* into the following *lambda-expression*.

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

Note that `f` 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*. I discuss qualified lookup below.

NOTE: I'm not sure if the return type should be `decltype(auto)` or `auto&&`. My feeling is that using `decltype(auto)` will lead to fewer surprises.

Similarly, the use of `operator>`, either as an argument or as the initializer of a variable would correspond to this lambda expression:

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

The proposed mechanism should not affect any existing overload resolutions. The *only* time this rule is engaged is when an overload set is being deduced against a parameter whose type is just plain `T`. This currently results in a deduction failure, and hence ill-formed programs.

Deductions and conversions involving *id-expressions* that name a single function are *not* affected by this feature.

Qualified lookup

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

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

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

Yields this lambda:

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

Operators

To support the shorthand for operators, we'll have to add a new category of *unqualified-id*. These ids simply refer to the generic lambdas that invoke the operator.

In general, the rules for synthesizing lambdas depend on the operator. For operators that have only a binary or unary form, we can synthesize the lambda directly:

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

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>
  decltype(auto) operator()(T&& x) const
  {
    return op std::forward<T>(x);
  }

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

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

The function call and index operators would have the following forms:

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

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

Operator functions

We should also be able to use *operator-function-ids* to name user-defined overloads. For example, the use of `operator>` here

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

would result in a lambda expression like this:

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

Note that I'm being lazy with the lambda here. The rules for synthesizing this lambda should be identical to those for `(>)`.

Forwarding calls to members

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 function call using the same *postfix-expression*.

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

Proposed wording

5.1.1 General [expr.prim.general]

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

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

operator-lambda-id:
  ( operator )
```

Add the following paragraph somewhere:

An *operator-alias-id* denotes a generic lambda that applies an set of operands to the operator. The *operator* in an *operator-alias-id* shall not be `new`, `new[]`, `delete` or `delete[]`. The corresponding lambda and its closure type depend on the operator:

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

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

- If the *operator* is `[]`, that *lambda-expression* is

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

- If the operator is one of `+`, `-`, `*`, or `&`, that expression is a prvalue object of unique, unnamed, non-union class type that is equivalent to

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

- Otherwise, that expression is the *lambda-expression*

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

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.

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)
{
  return f(std::forward<decltype(args)>(args)...);
}
```

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

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

- 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 — not even close. 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 his comments and corrections on an early draft of this document.