Overload sets as function arguments

Andrew Sutton <asutton@uakron.edu>

Date: 2015-09-25

Document number: P0019R0

Introduction

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)
{
  transform(first, last, f);
}
```

Unfortunately, this doesn't work if `f` names overloaded function or a function template. That's too bad because is the clearest possible expression of my intent, and because it happens to work when `f` names a single function (of the appropriate type).

Instead, I need to use a lambda algorithm with a concrete argument in order to get the compiler to select an appropriate overload of `f`.

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

This works, but it's not as clear and concise as it could be. I would much prefer to write the former.

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:

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

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

```
auto gt = operator>;
```

This feature can be provided without introducing runtime overhead.

How it works

The mechanism that makes this language feature work is to synthesize a lambda expression whenever an overload set is named. In this example:

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

The *id-expression* `f` (assuming it names an overload set) would correspond to the following lambda expression:

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

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

Note that this transformation described below doesn't work for unary operators. We would need a mechanism to select between a unary and binary operator when the lambda is instantiated. For such operators, we could synthesize a polymorphic function object:

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

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

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

Proposed wording

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

Editor's 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 unqualified *identifier* `f`, that expression is the *lambda-expression*:

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

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

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

- However, if `E` is an unqualified *operator-function-id*, of the form `operator@`, the lambda closure type depends on the operator:

  – If the *operator-function-id* is `()`, that expression is the *lambda-expression*

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

  – Otherwise, if the operator is one of `[]`, that expression is the *lambda-expression*

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

  – Otherwise, 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
      {
        return @ std::forward<T>(x);
      }

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

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

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

- Otherwise, the program is ill-formed.

Issues

- The proposal is missing synthesis rules for pre/post-increment and decrement.
- The wording does not currently allow for qualified operator names.
- The current proposal does not support for *conversion-ids* or
- The language mechanism requires the use of a library function. It would be better if there there were a term form "the forwarded expression", or possibly language support to simplify forwarding (e.g., `fwdexpr(e)`).

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);
```

I believe 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.