Implicit Evaluation of "auto" Variables and Arguments

Document number: N4035 (update of N3748)
Authors: Joël Falcou, University Paris XI, LRI
Peter Gottschling, SimuNova
Herb Sutter, Microsoft
Date: 2014-05-23
Project: Programming Language C++, Evolution Working Group
Reply to: Peter.Gottschling@simunova.com

Revision to N3748

1. Prioritized syntax changed.
2. Third alternative introduced.
3. Library-based disabling introduced.
4. Some type traits removed.

Altogether, the proposal is a clear simplification to its predecessor.

1 Motivation

Type detection for variables from expressions' return type:

```
auto x= expr;
```

has proven high usability. However, it fails to meet most users' expectations and preferences when proxies or expression templates (ET) are involved, e.g.:

```
matrix A, B;
// setup A and B
auto C= A * B;
```

Many people would expect C to be of type matrix as well. Whether C is a matrix depends on the implementation of **operator**$*$. In the case that the operator returns a matrix then C is a matrix.

For the sake of performance, computationally expensive operators very often return an *Expression Template* and delay the evaluation to a later point in time when it can be performed more efficiently—because we know the entire expression and where the result is stored so that we can avoid the creation and destruction of a temporary. The impact of this approach to the beforementioned example is that C is not of type matrix but of some intermediate type that was probably only intended to be used internally in a library.

Moreover, we assume that even people who are aware of expression templates will very often prefer the evaluated object (here a matrix containing the product of A and B) and not an unevaluated or partially evaluated object (representing the product of A and B).

We therefor need a mechanism to evaluate objects of certain types implicitly and sufficient control over this mechanism.

2 Goals

The implicit evaluation shall:

1. Enable class implementers to indicate that objects of this class are evaluated in an **auto** statement;
2. Enable them to determine the type of the evaluated object;
3. Enable them to implement the evaluation;
4. Allow programmers to explicitly disable this evaluation;
5. Provide information about the return type;
6. Allow for efficient use as function arguments;
7. Establish a compact and intuitive notation; and
8. Maximize backward compatibility.

3 Example

For the sake of illustration, we start with a typical implementation of expression templates:

```
class product_expr; // forward declaration

class matrix
{ ...
  matrix& operator=(const product_expr& product)
  { /* perform matrix product here */ }
};

class product_expr { ... };

inline product_expr operator*(const matrix& A, const matrix& B)
{ return product_expr(A, B); }

int main()
{
  matrix A, B;
  // setup A and B
  auto C= A * B; // type of C is thus product_expr
}
```

The **auto** variable C yield the type of the assigned expression which is in this case product_expr not matrix. We could have written:

```
matrix C= A * B;
```

and C would obviously be a matrix and also contain the *evaluated* product.

In the example above, the anonymous object product_expr(A, B) apparently *represents* the product of A and B and the product of two matrices is a matrix as well. Thus, to meet the users' typical expectations/preferences, we need to express how we create from an object *representing* A∗B an object that actually *is* A∗B.

4 Solution

To achieve the goal that the type of C becomes matrix we know of three approaches:

- Operator notation;
- **using** declaration; and
- Specialization of decay.

In discussions in the Chicago meeting and the reflector, the **using** declaration found the strongest consensus. We therefore, focus on this one and mention the alternatives later

4.1 Preferred Approach

It was suggested in the reflector to introduce an **operator auto** to enable this implicit evaluation. In the example above, we would expand the implementation of product_expr:

```
class product_expr
{
public:
  product_expr(const matrix& arg1, const matrix& arg2)
    : arg1(arg1), arg2(arg2) {}

  using auto= matrix;

private:
  const matrix &arg1, &arg2;
};
```

The actual evaluation can be implemented in both classes:

- Either in matrix with a constructor accepting product_expr; or
- In product_expr with a conversion operator towards matrix.

4.2 Alternative Approaches

Operator notation: The idea was to introduce a new operator like:

```
class product_expr
{
  ...
  matrix operator auto() { ... }
};
```

The advantage of this form is that the implicit evaluation could be implemented independently from constructors and conversion operators. However, such generality seems to be rarely necessary according to current experience and feedback. Furthermore, the syntax of the conversion operator with return type deduction is so close that a lot of confusion would be expected.

decay: Arno Schödl suggests using specialization of decay. The type trait decay reflects the type behavior of **auto** variables, i.e. **typename** decay<**decltype**(expr)>::type yield the type of variable x in **auto** x= expr;. He suggests to turn the semantics around and define **auto** x= expr; as:

```
typename std::decay<decltype(expr)>::type x=expr;
```

In this case, the type of the implicit evaluation can be customized by specializing std::decay.

4.3 Disabling the Implicit Evaluation

In several situations, the programmer will need the unevaluated object and still likes to use the automatic type deduction. For this purpose, we propose denoting the declaration with the keyword **explicit**:

```
explicit auto D= A * B;
```

Daveed Vandevorde remarked that the evaluation could also be disabled on the library level:

```
auto D= noeval(A * B);
```

where noeval wraps the expression (for later unwrapping):

```
template <typename T>
struct noeval_type
{
    const T& ref;
    noeval_type(const T& ref) : ref(ref) {}
    operator T const&() { return ref; }
    using auto= T const&;
};

template <typename T>
auto noeval(const T& ref)
{
    return noeval_type(ref);
}
```

We have no strong preference to any of the two approaches.

4.4 Reflection

To refer to the return type of the **operator auto**, we propose the following template alias:

```
template <typename T>
using auto_result_type= ...;
```

When no **operator auto** is define the template alias returns T. For the types from this proposal it would be therefor:

Type expression	Result
auto_result_type<product_expr>	matrix
auto_result_type<matrix>	matrix

5 Backward Compatibility

The implicit evaluation only applies on types that are equipped with an **using auto** declaration and existing code is not affected.

6 Consistency

No matter what semantics we choose, we do want to end up where these cases are consistent:

```
// case 1: local variable
auto x = expr;

// case 2: function parameter
template<class T> void f( T x );
f( expr );

// case 3: lambda parameter
auto f = []( auto x ) { };
f( expr );
```

Today these are identical (except only that case 1 can deduce initializer_list, and there is some pressure to remove that inconsistency). We believe these should stay identical.

We propose that all type-deduced variables and parameters with value semantic are subject to implicit evaluation in the same way. All forms of references are not implicitly evaluated.

7 Summary

We proposed a user-friendly method to deal with expression templates and proxies for local variables by introducing an implicit evaluation.

8 Wording

To be done!