Implicit Evaluation of "auto" Variables

Document number:	P0672R0	
Date:	2017-06-18	
Revises:	N4035	
	N3748	

Authors:	Joël Falcou	University Paris XI, LRI
	Peter Gottschling	SimuNova
	Herb Sutter	Microsoft

Project:	Programming Language C++, Evolution Working Group

Reply to:	Peter.Gottschling@simunova.com

Revision to N4035

1. Expanded motivation.
2. No implicit evaluation for function arguments.
3. Implementations mentioned.

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. As Scott Meyers points out, many proxy types are designed to operate invisibly to emulate the behavior of the type they emulate (let's call it object type for distinction's sake in this document). Therefore, users should normally work with object type and only when explicitly requested with the proxy.

Type deduction with proxies can lead to errors and inefficient programs:

- Undefined behavior,
- Unsupported operations, and
- User-defined variable types requiring extra resources.

We will demonstrate this in the following sub-sections. In addition, we would like to mention its usage in another proposal.

1.1 Undefined Behavior

Scott Meyers illustrates in Item 6 of his book "Effective Modern C++ " that using **auto** with vector<**bool**> can lead to undefined behavior:

```
Widget w;
vector<bool> features(const Widget& w);

auto highPriority= features(w)[5];

processWidget(w, highPriority);
```

The problem with this code snippet is that features returns a temporary vector<**bool**> and highPriority can contain a stale reference to that temporary. Storing features(w)[5] into a **bool** variable does not cause this problem. This issue might not be detected by the compiler and the undefined behavior could be undiscovered in applications.

1.2 Unsupported Operations

In the following example we consider operations on matrices:

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

This code snippet suggests that C is a matrix containing the product of A and B. However, this depends on the implementation of the product. In case the **operator**∗ returns a matrix then C is a matrix and the program should compile (and run properly).

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 only intended to be used internally in the library. A typical implementation of an ET representing a product will not provide an **operator**+= so that the last line will not compile.

1.3 Inefficient Type Declaration

Sticking with the matrix example, a linear algebra library (esp. a generic one) will not come with a single matrix type but with a whole spectrum thereof. Thus, the result type of a mixed-type operation is not always obvious:

```
compressed_matrix<double, column_major, referring, ...> A;
block_compressed_matrix<float, owning, dim<3, 4>, ...> B;
...
optimal_matrix_type C= A * B;
C+= D;
```

In addition to the typing effort, the user has to determine the optimal matrix type for C. Well, there might not be a perfect type for the result but we can assume that our ambitious library author spends some time on this question and is willing to provide his experience to its users. He can provide a type trait for this purpose.

1.4 Idea

The problems in the preceding examples can be solved by declaring the (object) type explicitly — although this might be difficult as well as shown in the last example. We like to empower the programmer of the proxy-like types to declare an appropriate type for the **auto** variables.

1.5 Other Usages

Hubert Tong and Faisal Vali utilize "using auto" in the first version of their proposal for "Smart References through Delegation" (P0352R0).

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 these types;
6. Establish a compact and intuitive notation; and
7. 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
  C+= D; // Won't compile since product_expr provides no operator+=
}
```

The **auto** variable C yields 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, product_expr(A, B) emulates a matrix and can be used in most situations where a "real" matrix is used in a non-modifying fashion. In our example, we try to modify C and fail. To help the user avoiding such problems we prefer C actually being a matrix containing A∗B instead of a proxy *representing* 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 and the Rapperswil meeting and in the reflector, the **using** declaration found the strongest consensus. We therefore, focus on this one and mention the alternatives later

4.1 Preferred Approach

Our preferred approach is the ability to declare the type of an **auto** variable.

```
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.

Therefore, we have:

```
auto C= A * B; // type of C is matrix
```

Since references can only refer to sub-types, the type transformation shall not apply to them, e.g.:

```
auto& C= A * B; // type of C is product_expr& (stale ref.)
auto&& C= A * B; // type of C is product_expr&&
```

4.2 Alternative Approaches

4.2.1 Operator Notation

The idea is 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, this syntax is so close to conversion operator with return type deduction that it might confuse many people.

4.2.2 decay

Arno Schödl suggested to use 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 Committee Feedback (so far)

The voting in Rapperswil clearly indicated a preference for our proposed "using auto" approach.

4.4 Disabling the Implicit Evaluation

In certain situations, the programmer might need the unevaluated object and still likes to use the automatic type deduction.

This can be easily provided at library level (suggestion of Daveed Vandevorde):

```
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);
}
```

Alternatively, we could denote it with keyword **explicit**:

```
explicit auto C= A * B; // C is product_expr
```

The voting in Rapperswil showed preference to the first option.

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 Function and Lambda Arguments

Type parameters in lambdas are also expressed by the keyword **auto**. This raised the question whether type substitution for value parameters should also apply the implicit evaluation:

```
// 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 );
```

The disadvantage of substituting value parameters' types would be a premature evaluation of proxies and expression templates in function calls.

The voting in Rapperswil strongly opposed to the idea that all type-deduced variables and parameters with value semantic are subject to implicit evaluation in the same way. We also propose to refrain from this implicit evaluation of function and lambda arguments.

7 Implementations

Daveed Vandervorte implemented the proposal shortly after the Rapperswil meeting and demonstrated on a toy example the desired behavior. Unfortunately, this EDG-based realization is not available anymore.

Faisal Vali also implemented the feature in Clang: `https://github.com/faisalv/clang/tree/using-auto`.

8 Summary

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

9 Wording

The wording will be provided when the preferred direction is clearly enough expressed by the committee.