Implicit Evaluation of "auto" Variables and Arguments

Document number:	N3748	
Authors:	Joël Falcou	University Paris XI, LRI
	Peter Gottschling	SimuNova
	Herb Sutter	Microsoft
Date:	2013-08-30	
Project:	Programming Language C++, Evolution Working Group	
Reply to:	Peter.Gottschling@simunova.com	

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 Solution

3.1 Implicit Evaluation

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.

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) {}

  matrix operator auto()
  {
    matrix R(num_rows(arg1), num_cols(arg2));
    R= *this;
    return R;
  }
private:
  const matrix &arg1, &arg2;
};
```

The **operator auto** in this example serves three purposes:

- It indicates that an assignment of a product_expr object to an **auto** variable causes an implicit evaluation.
- The **auto** variable will have type matrix.
- The operator also implements the evaluation.

Thus, in the presence of this operator, the **auto** variable C will have type matrix and contain the product. (Depending on the matrix implementation, one could write the operator in one line but this is a secondary detail here.)

3.1.1 Default Implementation

In the previous listing, the evaluation was realized by an assignment. Usually, there is an equivalent constructor, e.g.:

```
class matrix
{ ...
  matrix(const product_expr& product)
  { /* construct a new matrix by performing a matrix product */ }
};
```

Such a constructor allows for short implementations of the **operator auto**, e.g.:

```
class product_expr
{ // ...
  matrix operator auto()
  {
    return matrix(*this);
  }
};
```

We propose considering this pattern as default implementation, i.e.:

```
class some_temporary
{ // ...
  result_type operator auto() = default;
};
```

corresponds to:

```
class some_temporary
{ // ...
  result_type operator auto()
  {
    return result_type(*this);
  }
};
```

Discussion: Alternatively the default implementation could be:

```
class some_temporary
{ // ...
  result_type operator auto()
  {
    result_type result;
    result= *this;
    return result;
  }
};
```

However, this requires that the result_type is default-constructable and that the temporary can be assigned to this default-constructed object (in the context of matrices, the assignment must be allowed to change the dimension of the target matrix). We believe that the primary implementation is more generally usable.

3.1.2 Alternative Syntax for the Operator

Distinction by signature: In the context of return type deduction of normal functions (N3638), another **operator auto** was proposed which deduces its type from the actual return statement. Although we are afraid that the combination of type deduction and conversion operator is very dangerous, the presence of this feature in the standard would suggest another syntax:

```
operator auto result_type();
```

Then the two kinds of **operator auto** would be distinguishable by their signature. However, the distinction is not particularly eye-catching and we are afraid that the two different meanings of **operator auto** could a common source of confusion.

Using declaration: Daveed Vandevorde proposed on the reflector to denote the automatic evaluation by '**using auto**' for better distinction from conversion operators. The example above would read:

```
class product_expr
{
  using auto= matrix;
  // ...
};
```

The actual evaluation can be implemented in both classes:

- Either in matrix with a constructor taking a single object of type product_expr as argument; or

- In product_expr with a conversion operator towards matrix.[1]

3.2 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;
```

The object D contains whatever is returned by **operator**∗ regardless of whether an **operator auto** exists or not, in our case the expression template product_expr(A, B). Creating a non-explicit **auto** variable from D would of course trigger the evaluation:

```
explicit auto D= A * B; // not evaluated
// now tune some parameters of D
auto E= D; // now the expression is evaluated
```

Avoiding the immediate evaluation in the first line allow clever programmers doing clever tricks with D by either twisting parameters of D or generating another expression template from D that can be evaluated more efficiently.

In the reflector discussion, Nevin Liber suggested denoting the non-evaluated expressions by &&:

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

Although the notation is more concise, we prefer the more intentional '**explicit**' keyword.

3.3 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

Open question: Can we define auto_result_type with **decltype**? Or is this a job for the compiler?

3.4 Implicit Evaluation of Function Arguments

When expression templates are passed to function templates the programmer needs the choice whether the argument is evaluated or kept as expression template. Keeping the ET as it is enables a whole universe of optimization by transforming the represented Abstract Syntax Tree (AST). On the other hand, the types for the ET provide usually a much more limited interface than the type

[1]A weird question is: when a conversion operator has a deduced type, can we then write: **using auto**= **auto**; ? To be fair, **auto operator auto**() would not be better. For short, the combination with automatic return type deduction in conversion seems inapt regardless of the notation.

of the evaluated expression so that many implementations are much simpler after evaluation (or possible at all). For instance, a print function would be rather cumbersome when implemented for product_expr and we probably prefer doing it in terms of matrix. We can do this by introducing an **auto** variable:

```
template <typename Expr>
void print(const Expr& expr)
{
  auto A= expr; // A should now be a container, e.g. matrix
  // operate on A
}
```

Here the first statement in this function assures that A is container with readily evaluated data whenever all expression templates are equipped with an **operator auto**. Conversely, all types without **operator auto** are assumed to be container and are just copied.

The drawback of this approach is that we always create a new object—even when expr is already a container. For the sake of efficiency, we want to avoid this surplus copy.

3.4.1 Examples

To this end, we suggest the template alias auto_or_ref that (usually) evaluates to the result type of **operator auto** if defined; otherwise to a reference of the argument type: This alias allows us to avoid the copy:

```
template <typename Expr>
void print(const Expr& expr)
{
  // A is a container but not copied if expr is already a container
  const auto_or_ref<Expr> A= expr;
  // operate on A
}
```

In this example, we have no surplus copy: ETs are passed by reference and evaluated in the function whereas containers are referred twice (which certainly can be optimized away by compilers).

However, there might be cases where even in the presence of an **auto operator** the evaluation creates an overhead. Say, we have a function on vectors like the cross product:

```
template <typename Vector1, typename Vector2>
inline auto cross(const Vector1& v1r, const Vector2& v2r)
{
    const auto_or_ref<Vector1> v1= v1r;
    const auto_or_ref<Vector2> v2= v2r;
    // ... multiple read accesses on v1 and v2
}
```

This function shall be called with different types of arguments:

```
auto z1= cross(v, w); // #1 two containers
auto z2= cross(v, A*w); // #2 expression template
auto z3= cross(A[iall][2], w); // #3 proxy for matrix column
```

The first case is that both arguments are containers and we simply want to refer them.

In the second case, we have a matrix vector product as expression template which represents a column vector. Accessing an element of this ET would require a product of the matrix's row with the vector. So, it would be less expensive to compute the resulting vector once and use its elements. The situation might be even worse when the ET doesn't even provide an element access, e.g. when the matrix is a column-wise compressed matrix (and providing a single result entry is as expensive as the complete operation). For short, expression templates should be evaluated in any case.

In the third case is the most difficult. We assume that the access to a matrix column is realized by a proxy (which provides an **auto operator**). Using the proxy directly might be more efficient than paying the price of creating and destroying a temporary vector. However, whether evaluating the proxy is beneficial might depend on the type of object that the proxy refers to and even on the platform the executable is running. We therefor suggest to provide the user the ability to decide when a type used as function argument is evaluated.

3.4.2 Default Behavior and Specialization

In order to enable user-defined behavior, we propose a two-step definition of auto_or_ref:[2] a type trait named auto_or_ref_trait that determines whether a type should be evaluated and the template alias auto_or_ref based on it:

```
template <typename T>
struct auto_or_ref_trait
  : conditional< !is_same<T, auto_result_type<T>>::value, true_type, false_type>::type
{};

template <typename T>
using auto_or_ref=
    typename conditional<auto_or_ref_trait<T>::value,
                         auto_result_type<T>,
                         T&>::type;
```

The first class determines the default behavior:

- For a type with an **auto operator** (i.e. when T is different from auto_result_type<T>), the trait class is derived from true_type indicating that the object will be evaluated.
- Opposed to it, types without **auto operator** are derived from false_type indicating that the object will only be referred.

Programmers that want their proxy type not evaluated have to write:

```
template <typename U>
struct auto_or_ref_trait<my_proxy_type<U>>
  : false_type
{};
```

Of course, the type can implement arbitrarily complicated conditions.

The template alias auto_or_ref will then be defined as either the evaluated type or a reference.

3.4.3 Mutable Arguments

We refrain from distinguishing constant and mutable references, i.e. introducing an alias "auto_or_const_ref" since we consider the choice between evaluating or referring orthogonal to constancy. When the reference is **const** then the object must not be modified in the algorithm, thus, we can also declare the evaluated object as **const**.

[2] N1489 argues that template aliases should not be specialized.

On the other hand, the mutable version:

```
template <typename Expr>
void weird_function(Expr& expr)
{
  auto_or_ref<Expr> A= expr;
  // modify A
}
```

makes not really sense (nonetheless we are open to learn counter-examples). Of course, we can safely modify both the reference and the evaluated object. However, in the latter case the object is only a temporary in the function and the impact would not be visible from outside.

4 Backward Compatibility

Programs containing statements like:

```
auto x= expr;
```

where expr is an expression template and x shall not contain the evaluated object need to be transformed into:

```
explicit auto x= expr;
```

after the type of expr was equipped with an **operator auto**.

Conversely, several programs that do not work today because the programmer expected an evaluated will work in the future after **operator auto** is introduced.

5 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.

The problem can be approached from two sides:

- Types with defined **auto operator** are always evaluated unless **auto** or T is decorated with **explicit**. This implies that **explicit** must be allowed for generic parameters of named functions and lambdas. The drawback of this approach is that function calls with evaluatable types as arguments would be even incompatible to C++98.

- Types with defined **auto operator** are only evaluated if **auto** or the template parameter is decorated with "implicit". The drawbacks of this notation is that the variable declaration with implicit evaluation would be more verbose and that we would introduce a new keyword. At least, this should not break existing code since free names are not allowed in front of **auto** or a template parameter.

Both approaches are not satisfactory and we still consider it an open question how to achieve this in the presence of implicit evaluation.

6 Summary

We proposed a user-friendly method to deal with expression templates and proxies for local variables by introducing an implicit evaluation. There are still some open questions for which we hope to find an answer soon.

7 Wording

To be done!