Guaranteed copy elision through simplified value categories

ISO/IEC JTC1 SC22 WG21
P0135R0
Richard Smith
richard@metafoo.co.uk
2015-09-27

This paper presents a case for guaranteeing that certain forms of copy elision always occur (in particular, when the source object is a temporary), and removing the semantic checks on the copy or move constructor in the case where the guaranteed elision occurs. The approach described herein achieves this not by eliding a copy, but by reworking the definition of the value categories (glvalue versus prvalue) such that it is unnatural to perform the copy in the first place.

Copy elision

ISO C++ permits copies to be elided in a number of cases:

- when a temporary object is used to initialize another object (including the object returned by a function, or the exception object created by a *throw-expression*)
- when a variable that is about to go out of scope is returned or thrown
- when an exception is caught by value

Whether copies are elided in the above cases is up to the whim of the implementation. In practice, implementations always elide copies in the first case, but source code cannot rely on this, and must provide a copy or move operation for such cases, even when they know (or believe) it will never be called.

This paper addresses only the first case. While we believe that reliable NRVO ("named return value optimization", the second bullet) is an important feature to allow reasoning about performance, the cases where NRVO is possible are subtle and a simple guarantee is difficult to give.

Why should copy elision be mandatory?

A recent thread on the std-proposals mailing list provides a long list of reasons why the current approach to copy elision is problematic. Here are some highlights:

- The language requires provision of a copy or move constructor for a type *even if* the programmer and the compiler agree that it will never be called. This means it is impossible or very difficult to write factory functions / "named constructors" for non-moveable types:

  ```
  struct NonMoveable { /* ... */ };
  NonMoveable make() { /* how to make this work without a copy? */ }
  ```

  As a result, programmers are forced to work around this limitation via dynamic memory allocation or similar.
- The performance difference between elision and move construction is not always negligible (particularly for large objects), but portable code cannot rely on elision

occurring today. As a result, some programmers still prefer to use out-parameters rather than simply returning by value.
- The "almost always `auto`" style has a big problem: the "almost". This is because there's no way to initialize a variable of a non-moveable type from an expression of that type.

```
auto x = make(); // error, can't perform the move you didn't want,
                 // even though compiler would not actually call it
```

- Allowing the execution semantics of a program to be up to the implementation harms portability and the ability to reason about the code.

What would mandatory copy elision look like?

```
struct NonMoveable {
  NonMoveable(int);
  NonMoveable(NonMoveable&) = delete;
  void NonMoveable(NonMoveable&) = delete;
  std::array<int, 1024> arr;
};
NonMoveable make() {
  return NonMoveable(42); // ok, directly constructs returned object
}
auto nm = make(); // ok, directly constructs 'nm'
```

Value categories

The approach we take to provide guaranteed copy elision is to tweak the definition of C++'s 'glvalue' and 'prvalue' value categories (which, counterintuitively, categorize *expressions*, not values). C++ currently specifies the value categories as follows:

> - An *lvalue* (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. - end example]
> - An *xvalue* (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). Certain kinds of expressions involving rvalue references (8.3.2) yield xvalues. [Example: The result of calling a function whose return type is an rvalue reference to an object type is an xvalue (5.2.2). - end example]
> - A *glvalue* ("generalized" lvalue) is an lvalue or an xvalue.
> - An *rvalue* (so called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.
> - A *prvalue* ("pure" rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue. - end example]

However, these rules are hard to internalize and confusing -- for instance, an expression that creates a temporary object designates an object, so why is it not an lvalue? Why is `NonMoveable().arr` an xvalue rather than a prvalue? This paper suggests a rewording of these rules to clarify their intent. In particular, we suggest the following definitions for glvalue and prvalue:

- A *glvalue* is an expression whose evaluation computes the location of an object, bit-field, or function.
- A *prvalue* is an expression whose evaluation initializes an object, bit-field, or operand of an operator, as specified by the context in which it appears.

That is: **prvalues perform initialization, glvalues produce locations**.

Denotationally, we have:
glvalue :: Environment -> (Environment, Location)
prvalue :: (Environment, Location) -> Environment

So far, this is not a functional change to C++; it does not change the classification of any existing expression. However, it makes it simpler to reason about why expressions are classified as they are:

```
struct X { int n; };
extern X x;
X{4};   // prvalue: represents initialization of an X object
x.n;    // glvalue: represents the location of x's member n
X{4}.n; // glvalue: represents the location of X{4}'s member n;
        //          in particular, xvalue, as member is expiring

using T = X[2];
T{{5}, {6}};    // prvalue: represents initialization of an array of 2 X's
T{{5}, {6}}[0]; // xvalue: represents location of expiring array element
```

Implications of refined value categories

Now we have a simple description of value categories, we can reconsider how expressions in those categories should behave. In particular, given a class type `A`, the expression `A()` is currently specified as creating a temporary object, but this is not necessary: because the purpose of a prvalue is to performs initialization, it should not be the responsibility of the `A()` expression to create a temporary object. That should instead be performed by the context in which the expression appears, **if necessary**. However, in many contexts, it is not necessary to create this temporary object. For instance:

```
// make() is a prvalue (it returns "by value"). Therefore, it models the
// initialization of an object of type NonMoveable.
NonMoveable make() {
  // The object initialized by 'make()' is initialized by the following
  // constructor call.
  return NonMoveable(42);
}
// Use 'make()' to directly initialize 'nm'. No temporary objects are created.
auto nm = make();

NonMoveable x = {5}; // ok today
NonMoveable x = 5; // equivalent to NonMoveable x = NonMoveable(5),
                   // ill-formed today (creates a temporary but can't move it),
                   // ok under this proposal (does not create a temporary object)
```

We conclude that a prvalue expression of class or array type should **not** create a temporary object. Instead, the temporary object is created by the context where the expression appears, if it is necessary. The contexts that require a temporary object to be created ("materialized") are as follows:

- when a prvalue is bound to a reference
- when member access is performed on a class prvalue
- when array subscripting is performed on an array prvalue
- when an array prvalue is decayed to a pointer
- when a derived-to-base conversion os performed on a class prvalue
- when a prvalue is used as a discarded value expression

Note that the first rule here already exists in the standard, to support prvalues of non-class, non-array type. The difference is that, with the proposed change, the language rules are now uniform for class and non-class types.

Alternatives

There appear to be two main alternatives to this proposal:

- **Do nothing**. We have lived with the current rules for a long time, and we can continue to do so. This means:
  - we have no guarantee of efficiency around temporaries
  - we need to provide copy / move constructors that we know will / should never be called
  - we don't have a good way to write factory functions for non-moveable typesk
- **Guarantee copy-elision in a different way**. It's not clear how we would do this. We would also miss the opportunity to make value categories easier to understand and learn.

The main advantages of those approaches over this proposal is that they retain the conceptual model that `Type(...)` creates a temporary object. However, that model is partially an illusion: it only explains the behavior when `Type` is a class type.

Acknowledgements

The author wishes to thank David Krauss and Jonathan Coe for their feedback on this proposal, and all the contributors to the std-proposals discussions on this topic for their ideas.