Title: Attribute [[asserts_rvo]]

Document: P3134

Date: 2024-02-15

Audience:

- SG14

Author:

- name: Nicolas Fleury email: nidoizo@gmail.com
- name: Patrice Roy email: patricer@gmail.com

Introduction

Looking solely at function's signature in C++, one does not know if the Return Value Optimization (RVO) will be performed when the function will be called: the ability to apply such optimizations in part depend in part on the function's implementation, and even carefully written code can change and evolve over time in such as way as to break RVO opportunities. This proposal aims to add the ability to know directly from a function signature that this function is written in a way that enables RVO.

Note: this paper is part of the wider effort described in P2966.

Motivation

Copy elision is not guaranteed in multiple scenarios:

```
MyClass SomeFunction1(MyClass myObj)
{
    ...
    return myObj;
}

MyClass SomeFunction2()
{
    MyClass obj1 = ...;
    MyClass obj2 = ...;
    return someCondition ? obj1 : obj2;
}

MyClass g_SomeGlobalObj;
MyClass SomeFunction3()
{
    ...
    return g_SomeGlobalObj;
}
```

```
struct MyOtherClass
{
    MyClass SomeFunction4()
    {
        ...
        return m_MemberObj;
    }
    MyClass SomeFunction5()
    {
        ...
        return m_StaticOb;
    }
    MyClass m_MemberObj;
    static MyClass m_StaticOb;
}

class MySubClass : public MyClass { ... };

MyClass SomeFunction6()
{
    MySubClass myObj;
    ...
    return myObj;  // this is not moving on multiple compilers; P1155 proposes to
move
}
```

Considering that, given the following two declarations, one could wonder which one programmers will prefer when in a context where performance is primordial and copying SomeClass is heavy:

```
SomeClass Foo();
void Foo(SomeClass&);
```

While the second declaration might look more archaic, it is often better to prevent unwanted copies. Indeed, whilst one can write `SomeClass Foo()` if RVO is guaranteed, the caller cannot know if the implementation is written in a way that allows RVO by just looking at the function's declaration. Even if one looks at the implementation of the function to confirm that copy elision can be done, that implementation can evolve over time in such as way that, eventually, copy elision is done no more:

```
MyClass SomeFunctionV1()
{
    MyClass obj;
    ...
    return obj;  // copy elision
}
```

```
MyClass SomeFunctionV2()
{
    if (SomeCondition())
    {
        static MyClass defaultObj = MyClass::MakeDefault();
        return defaultObj;
    }
    MyClass obj;
    ...
    return obj;  // no more copy elision
}
```

One could disable the copy constructor in `SomeClass`, but even that is not always possible:

```
std::vector<SomeClass> Foo();
void Foo(std::vector<SomeClass>&);
```

In some cases, even moving is costly, so one can really want RVO to occur:

```
std::array<SomeClass, 16> Foo();
```

This proposal seeks to make it possible to see from a function's declaration that neither a copy nor a move will ne done when calling that function:

```
[[asserts_rvo]] std::vector<SomeClass> Foo();
```

Naming and other considerations

Naming

We considered simply naming this attribute `[[rvo]]` but after some thought, we are proposing `[[asserts_rvo]]` to clarify the fact that it does not affect RVO itself, but only asserts it will done if the function is called appropriately.

The attribute name `[[asserts_rvo]]` implies a contract for the function implementation itself, and not its usage.

Validating external code

We could consider the same attribute on calling site:

```

```

```
SomeClass ExternalFoo();
...
[[asserts_rvo]] SomeClass someObj = ExternalFoo();
```

This is not the core of this proposal, and the usage is much more marginal, but it is worth mentioning.

Summary

The attribute `[[asserts_rvo]]` would guarantee that the function implementation can and will do RVO optimization if called in such a way as to enable this optimization. If not, the compiler would emit a compilation error.

The attribute is for two use cases:

- Annotating the function's declaration, so that programmers can call it trusting it's not doing an heavy copy.
- Checking the function's definition to enforce that RVO can be done, and ensure it does not get broken with code evolution.

Like `inline`, the attribute can be for both declaration and definition, optional for the other when specified for one.