Document Number: P0028R0
Date: 2015-09-15
Reply to: josedaniel.garcia@uc3m.es

Using non-standard attributes

J. Daniel Garcia
Computer Science and
Engineering Department
University Carlos III of Madrid

Luis M. Sanchez
Computer Science and
Engineering Department
University Carlos III of Madrid

Massimo Torquati
Computer Science Department
University of Pisa

Marco Danelutto
Computer Science Department
University of Pisa

Peter Sommerlad
Institut für Software
HSR Rapperswil

1 Introduction

This paper proposes a new attribute to take control on attributes namespace in order to avoid the need of repetitive use of attributes namespaces.

Attributes [1] provide a useful way to add annotations to source code with implementation defined effects. Implementations are expected to add their own attribute namespace where their attributes are defined. In fact, scoped attributes —those under a specific namespace— are specified as conditionally supported. While this approach provides a clean way for different implementations to add their own attributes, it may lead to very verbose code.

To better support the introduction of conditionally supported attributes we propose the introduction of a new standard attribute, the `using` attribute, to avoid repetition of attribute namespaces when making extensive use of attributes to perform code annotations.

2 Problem

Attributes have proved to be a very useful way to perform source code annotations. One example of this is the set of attributes [2] defined in the context of the *REPARA* project (`http://www.repara-project.eu`).

A simple example of such use is the annotation of computational kernels that can be later transformed to different programming models.

```
void f() {
  [[rpr::kernel]]
  for (int i=0; i<iterations; ++i) {
    do_something();
  }
}
```

However, in complex cases multiple attributes need to be used in a single annotation. This results in a verbosity that will make most implementations to look for very short attribute namespaces names.

```
void f() {
  [[rpr::pipeline(bound, 8, blocking), rpr::stream(A,B)]]
  for (int i=0; i<iterations; ++i) {
    [[rpr::kernel, rpr::out(a), rpr::target(cpu) ]]
    a = get_value();

    [[rpr::kernel, rpr::farm(4,ordered), rpr::in(A,C), rpr::out(A,B), rpr::target(cpu,gpu)]]
    for (int j=0;j<max;++j) {
      b = f(a,c);
    }
    [[ rpr :: kernel , rpr :: in (A,B)]]
    g(a,b);
  }
}
```

An alternate solution could be to combine multiple attributes with a more complex syntax, but this would introduce complexities in the attribute syntax itself while making worse the ability to understand the annotations.

3 Proposal

We propose a new standard attribute to introduce an attribute namespace into the global scope for attributes. Other scoping options are discussed in section 3.3.

```
// Current situation
void f() {
  [[ rpr :: kernel , rpr :: target (cpu,gpu)]]
  do_task ();
}

// Proposed change
void g() {
  [[ using(rpr), kernel , target (cpu,gpu)]]
  do_task ();
}
```

In this paper we have named the proposed attribute `using` for the similarity to using directives. However, this choice is for exposition only and a different name could be used.

3.1 Effect of the using attribute

The effect of a `using` attribute is to introduce all the attributes names from a specific attribute namespace to the global attribute namespace. Thus, after a `using` attribute, all the attributes from that namespace can used without explicit mention to the namespace.

```
void g() {
  [[ using(rpr), kernel ]] // equivalent to [[rpr :: kernel]]
  do_task ();
```

As a generalization, it should be possible to introduce multiple attribute namespaces in a single attribute specifier.

```
void g() {
  [[ using(rpr,optimization), kernel , unroll (4)]]
  for (int i=0; i<max; ++i) {
    do_task(i );
  }
}
```

3.2 Name lookup

The introduction of the `using` attribute requires additional rules when an attribute specifier is seen. The question that needs to be answered is how an attribute name is resolved.

If an attribute specifier contains a non-scoped attribute name, it is first checked against the list of standard attributes. If it is found, the attribute lookup is finished.

If a non-scoped attribute is a non-standard attribute, it is checked against the attribute name lists of all the attribute namespaces that have been introduced through an `using` attribute. The outcome of this process could be one of the following:

1. The lookup produces exactly one match. The program is considered to be well formed.
2. The lookup produces more than one match in different attribute namespaces. The effect, in this this case, is implementation defined

3. The lookup produces zero matches. The attribute is ignored and the program is well-formed, although a diagnostic may be optionally emitted.

3.3 Scope of the using attribute

There are several design alternatives on the effect of a `using` attribute in terms of scoping:

- Effect within the current attribute specifier.
- Effect until the end of the current translation unit.
- Effect within the current scope.

3.3.1 Sequence specifier scope

The simplest implementation is to make effect of a `using` attribute limited to the current sequence specifier.

```
void f(X & x) {
  [[rpr::kernel, rpr::target(gpu), rpr::out(x)]] g1(x); // OK
  [[using(rpr), kernel, target(gpu), out(x)]] g2(x); // OK
  [[using(rpr), kernel]] [[target(gpu)]] g3(x); // Wrong. Target in different attr-specifier
}
```

This option is simple to implement and provides some usefulness to the original problem. For illustration we reproduce here our original pipeline example.

```
void f() {
  [[using(rpr), pipeline(bound, 8, blocking), stream(A,B)]]
  for (int i=0; i<iterations; ++i) {
    [[using(rpr), kernel, out(a), target(cpu) ]]
    a = get_value();

    [[using(rpr), farm(4,ordered), in(A,C), out(A,B), target(cpu,gpu)]]
    for (int j=0;j<max;++j) {
      b = f(a,c);
    }

    [[using(rpr), kernel, in(A,B)]]
    g(a,b);
  }
}
```

This option makes easier to use multiple attributes from the same attribute namespace. However, we still need to start with `using` every attribute specifier.

3.3.2 Translation unit scope

A second option, is that a `using` attribute has effect from the point in the source code where it is located to the end of the current translation unit.

This would allow to state a `using` specifier only once.

```
void f() {
  [[using(rpr)]]
  [[kernel,target(cpu)]] do_task1(); //rpr::kernel, rpr::target
  [[kernel,target(gpu),out(a)]] // rpr::kernel, rpr::target, rpr::out
  for (int i=0; i<max; ++i) {
    a[i]=do_task2();
  }
}
```

However, there are several cases where the user could be easily surprised.

```
void f() {
  [[using(rpr), kernel, target(cpu)]] // rpr::kernel, rpr::target
  do_task1();
}
```

```
void g() {
  [[ kernel ]]
  do_task3(); //rpr::kernel?
}
```

This situation could be made much worse by inserting `using` attributes in header files (e.g. inside an `inline` function). Consequently, we do not recommend this second alternative.

3.3.3 Current scope

A third option would be to use regular scoping rules for C++ lookup. That is a `using` attribute has effect from its point of use until the end of the current scope.

```
void f() {
  [[ using(rpr), kernel, target(cpu)]] // rpr::kernel, rpr::target
  do_task1();
}

void g() {
  [[ kernel ]]
  do_task3(); // rpr::kernel not found!
}
```

We think this is the most advisable solution. However, we recognize that it introduces additional complexities as it makes necessary to mix regular namespace management with attribute namespace management. Specific cases where complexities could arise are attributes in namespace scope or even in the global scope which are unavoidable when function definitions are attributed.

One limited version of this approach could be to allow scope based `using` attributes at function body scope only.

However for the sake of simplicity, we recommend the first option (sequence specifier scope).

3.4 Why a standard attribute?

It could be argued that if an extension needs an attribute as `using` it could add as part of the extension.

```
void f() {
  [[ rpr::using, kernel, target(cpu)]] // rpr::kernel, rpr::cpu
  do_task1();
}
```

However, this would be a non conforming extension as an implementation could see those unknown attributes in the global namespace.

Besides, we think that the proposed `using` attribute could be of utility to multiple implementations defining several attributes in the same namespace.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/20072013) under grant agreement n. 609666.

References

[1] Jens Maurer and Michael Wong. Towards support for attributes in C++. Working paper N2761, ISO/IEC JTC1/SC22/WG21, September 2008.

[2] Luis M. Sanchez et al. Static Partitioning Tool. Technical Report D3.3, REPARA Project, December 2014.