Document Number: P0028R1
Date: 2016-02-12
Reply to: josedaniel.garcia@uc3m.es
Audience: Evolution Working Group.

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

Changes

Changes from P0028R0 [1]:

- We now propose a namespace introducer instead of the previous *using* attribute.
- Scoping is limited to the current attribute list.

1 Introduction

This paper proposes a new attribute introducer to avoid the need of repetitive use of attributes namespaces.

Attributes [2] 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 addition of an attribute introducer, 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 [3] 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 attribute namespace introducer, to introduce an attribute namespace in the current attribute specifier.

```
// 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 introducer `using` for the similarity to using directives. However, this choice is for exposition only and a different name could be used. Possible alternate introducers could be: `namespace`, `use`, `with`.

3.1 Effect of the using introducer

The effect of a `using` introducer is to introduce all the attributes names from a specific attribute namespace into 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();
```

3.2 Name lookup

The introduction of the `using` introducer requires additional rules when an attribute specifier is seen.

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` introducer. 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 issued.

3.3 Scope of the using introducer

The effect of a `using` introducer is limited to the attribute list where it appears.

```
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-list
}
```

Acknowledgments

This proposal has been improved thanks to discussions with the following individuals: David Vandevoorde, Bjarne Stroustrup, and Michael Wong.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007–2013) under grant agreement n. 609666.

References

[1] J. Daniel Garcia, Luis M. Sanchez, Massimo Torquati, Marco Danelutto, and Peter Sommerlad. Technical report.

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

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