Problems with reference_closure

Document number: N2830=09-0020
Date: 2009-02-04
Authors: John Spicer, Daveed Vandevoorde, Douglas Gregor
Reply-to: John Spicer (jhs@edg.com)
Project: Programming Language C++, Core Working Group

Introduction

The lambda proposal as voted into the working draft requires that if a lambda captures all of its local variables by reference, the closure class must be derived from std::reference_closure. This feature was provided to allow the user to write functions that can take as arguments any suitable lambda.

Now that we have additional implementation experience, we believe that the requirement that lambdas derive from std::reference_closure is over-specification, does not provide additional optimization opportunities, can in fact constrain implementations from certain optimizations, and ends up providing two incompatible mechanisms for passing lambdas to other functions.

std::reference_closure vs. std::function

The working draft already provides a powerful, general-purpose mechanism that can be used to pass lambdas to functions: std::function. std::reference_closure provides a limited subset of the functionality of std::function and will result in the use of two incompatible mechanisms, complicating interfaces and reducing interoperability. In his national body comments, Doug Gregor wrote:

> std::reference_closure is a premature optimization that provides a limited subset of the functionality of std::function intended to improve performance in a narrow use case. However, the “parallel application performance” benchmark used to motivate the inclusion of std::reference_closure was flawed in several ways:
>
> - it failed to enable a common optimization in std::function (implemented by all vendors), exacting a large and unrealistic penalty for copying std::function instances, and
> - it failed to account for parallel scheduler overhead or realistically-sized work units, both of which would dominate the costs measured by the benchmark in any realistic application.

Over-constraining Implementations

Most lambdas that would be required to derive from std::reference_closure will not in fact ever be passed to a routine that takes a std::reference_closure. The requirement that such lambdas derive from std::reference_closure actually makes it more difficult to optimize some common cases. A reference-only lambda requires at most a single pointer to access the stack information, but std::reference_closure forces an implementation to use two pointers. That penalizes the very common case where a lambda is passed as a function object to a template, for example:

```
void f0x(std::vector<int>& v) {
   sort(v.begin(), v.end(),
        [](int x, int y) -> bool { return x > y; });
}
```

The requirement that the lambda derive from std::reference_closure turns what would be an empty class into something with two pointers. That could, conceivably, make the use of the lambda slower than today's function-object version:

```
void f98(std::vector<int>& v) {
   sort(v.begin(), v.end(), std::greater<int>());
}
```

Without the std::reference_closure requirement, all of the uses of a given lambda are always within a given translation unit (including template instantiations from that translation unit). This means that a given implementation is free to implement lambdas in any way it chooses provided it follows the working paper definition using the “as if” rule. With std::reference_closure, however, the implementation details of the reference closure mechanism become part of the ABI. This means that implementations can’t do better than what is specified in the ABI. It also means that some implementation techniques (such as generating C code as the output of a C++ compiler) will probably not be able to generate code that is compatible other compilers.

Existing implementations

Of the four implementations that we know of, none has implemented this feature. This suggests that the feature presents implementation challenges and lacks compelling demand from users (or that vendors can satisfy that demand without using a feature like this).

Working Paper Changes

Remove 5.1.1 [expr.prim.lambda] paragraph 12.
Remove 20.7.18 [func.referenceclosure] (and its subsections).