J16/00-0063
WG21/N1286
November 7, 2000
J. Stephen Adamczyk
jsa@edg.com

The point of destruction of a call argument temporary

Introduction

I was recently trying to determine what the standard says about the point at which a temporary created to pass an argument to a call is destroyed. After a bit of work, I think I figured it out, and I think what the standard says is probably okay. However, because the rules are a bit complicated and surprising, I decided to write up my understanding so that there is a summary that will make things clearer for implementors, and to see whether (a) others agree with my conclusions, (b) others feel the specification should be changed, or (c) there's a desire to see this made clearer in the standard.

What's being created?

More on this in a moment, but the simple summary is that, when an argument is passed by value for a parameter of class type, the parameter is initialized by calling a copy constructor to copy the argument. This creates a local class object in the parameter. When the function returns, this class object must be destroyed. But who does this destruction, and when is it done?

```
struct A {
  A();
  A(const A&);
  ~A();
} a;
void f(A);
int main () {
  f(a);  // a is copied to the parameter; the parameter object is
         // later destroyed
}
```

Inside or Outside?

The first question I wanted to answer was whether the caller or the called function must do the destruction. There is existing practice for both approaches: some compilers call the destructor at the point of call, and some generate code within the called function to destroy the parameter on return[1]. Destroying the parameter within the called function has a certain appeal, because it allows a compiler to emit a single copy of the destruction code rather than a copy at each point of call. However, as it happens, 5.2.2 [expr.call] paragraph 4 makes it pretty clear that that is not allowed::

> The initialization and destruction of each parameter occurs within the context of the calling function. [Example: the access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a function-try-block (clause _except_) with a handler that could handle the exception, this handler is not considered.]

Under the “as if” rule, one could still put the destruction code inside the called function as long as there’s no way to tell the difference. One would have to do the parameter destructions after leaving any function-try-block, after leaving the range of the exception specifications of the called function, after evaluating any expression on the return statement, and, for functions that return a class type by value, after calling the copy constructor to copy the return value. But that’s implementation; from the point of view of the standard, it’s clear that the destruction is done by the caller. The question then remains: when does the destruction get done?

Is it a Parameter or a Temporary?

The class object constructed: what is it? Is it the parameter or is it a temporary? Well, sometimes it’s the parameter, and sometimes it’s both.

When the argument expression has the same type as the parameter (or a cv-qualified version or derived class thereof), no temporary is created. The argument expression is copied to the parameter via a copy constructor. In that case, it’s clear that the object created is the parameter and that it must be destroyed at the point of return of the function, as indicated by another sentence in 5.2.2 [expr.call] paragraph 4:

> The lifetime of a parameter ends when the function in which it is defined returns.

When the argument expression does not have the same type as the parameter (or a related type, as defined above), the initialization is really

- a temporary is initialized from the argument expression, and then
- the parameter is initialized by copying the temporary using a copy constructor.

In practice, implementations will usually elide the second step. If that is done (see 12.8 [class.copy] paragraph 15), the object created is both the parameter and the temporary, and as such it is destroyed at the later of the two points where those objects would be destroyed, which is to say, at the end of the full expression containing the call.

Note that the point of destruction for the two cases is different. One might be tempted to say that the standard should be changed to say that a temporary created to pass an argument by value is destroyed on return from the call, thus making those two cases the same. However, if we go a little further and consider a case like the second case except that the argument is passed by reference,

---

1. Note that all implementations call the constructor at the point of call. This is necessary because the constructor is selected by overload resolution, and might be different on different calls.

we find that in that case the class object that's created is the temporary alone, and as such has lifetime to the end of the full expression (12.2 [class.temporary] paragraph 5 mentions this case explicitly).

An example might be helpful:

```
struct A {
  A();
  A(const A&);
  ~A();
} a;
struct B {
  operator A();
} b;
void f(A);
void g(const A&);
int main() {
  f(a);  // (1) Object destroyed on return from function
  f(b);  // (2) Object destroyed at end of full expression
         //     (if no elision done, parameter destroyed on
         //     return, temporary destroyed at end of full
         //     expression)
  g(b);  // (3) Object destroyed at end of full expression
}
```

Changing the rules for (2) to make it like (1) would make (2) unlike the very similar (3). It suppose one could consider changing the rules for (3), but that starts to get pretty far from the original issue and might have side effects elsewhere.

In this example, of course, "on return" and "at end of full expression" are indistinguishable, but in general they are not the same. If the expressions were more complicated, for example

```
  h(f(b), g(b));
```

one could observe the difference between the two by way of the point of occurrence of any side effects of the destructor calls.

Conclusion

The rules in the standard are clear (with careful reading). They're a bit convoluted and they might appear inconsistent in some ways, but in spite of that they're probably reasonable. I recommend that we explore putting more text and examples into the standard to clarify the current definition, but that we not otherwise change it.