R1

A Discussion of Default argument instantiation

Erwin Unruh
12.03.97
Doc No. 97-0024R1, N1062R1

On Tuesday we had the discussion of what the core solution should look like. Here is the outcome:

At present clause 14 does not say anything about whether and when default argument expressions of template function specializations are instantiated. There may have been discussion and solutions in CoreIII, but they did not get into the WP. What we do have in the WP are:

14.7.3/3: An explicit specialization is not allowed to have default arguments, so we do not have to bother about them.
8.3.6/4 specifies a rule for default arguments of templates, but that is not important for our issue.

14.7.1/6 and 14.8.3: A declaration is instantiated if it is used as a candidate function. Since this also happens if the function is not going to be called, we decided to leave this rule alone.

14.6.4.2: This was introduced to allow an instantiation take place using the debugger symbol table at runtime. No wording in there exists which precludes it being applied to default arguments as well. So if we fix a point where a default argument should be instantiated, this paragraph automatically allows an implementation to use the global program as a substitute.
The words about *with external linkage* have been noticed by coreIII to be a problem. A solution to them should affect both the function body and default arguments.
Applying these rules to default arguments in the present state may disallow the use of static functions in default arguments, but that is a very small price to pay and I am very willing to do that.

What also occurred to us is that we need the understanding that the declaration of a template function creates a definition context for that function. This is the context where the phase 1 name binding occurs and that is done both for function definitions as well as function declarations. This is a problem independent of the topic at hand.
I do not know whether we need wording in the WP to clarify this.

When explicitly instantiating a template default arguments are not affected. An explicit instantiation should only affect the body of a function or the initializer (maybe including constructor call) of a static data member. Any future work on explicit instantiation should take this into account.

During the week I realized that every template function specialization has several instantiation points even within a singe translation unit. Every use of the function creates one instantiation point.
So we do not have to worry about the binding being in a different place than that for the function.
Wherever a default argument is instantiated, the instantiation of the function body is allowed as well.

In order to fix the problem with the default arguments I think we need to do the following:

- add wording to say: a default argument is instantiated if and only if it is used.
- add wording to clarify: Where is the phase two name binding for a default argument.
- add an example similar to those used in the standard library.

This is a second cut for the wording. Thanks to Martin for the first cut, I took some of his words.

Proposed WP changes (words in italics still need work):

add to 14.7.1 *either as part of paragraph 1 or between 1 and 2 or between 2 and 3*

A *default argument expression specialization* is implicitly instantiated *only* when the function specialization is referenced in a context that requires the default argument expression to exist.

Add to 14.6.4.1 after paragraph 8:

The point of instantiation of a *default argument expression specialization* is the same as that for the function.
[Note: Even if only some of the calls use a default argument, all points of instantiation of the function can be used to instantiate the *default argument expression specialization.*]

Add the example somewhere:

```
template <class T> void foo( T t1, T t2 = T() );

class A { A(); } a;
class B { B(int); } b = 5;

foo(a,a); // #1
// fine. the body may be generated here, but the default argument isn't

foo(b,b); // #2
// fine, t2 = B() not checked here because not used.

foo(a); // #3
// fine. the default argument is instantiated here.

foo(b); // #4
// error, the default argument needs instantiation but B does not have a default constructor
```