Document Numbers: X3J16/96-0214
WG21/N1032
Date: November 13, 1996
Reply To: Bill Gibbons
bill@gibbons.org

Core-III Working Paper Changes (Motions and Editorial)

==

1. Clarify the semantics of "uncaught_exception".

(Rationale: it is safe to throw an exception from unexpected (because the exception is considered handled at entry to unexpected), but not from terminate (because terminate may not exit.) Since uncaught_exception is intended to indicate whether it is safe to throw an exception, the semantics associated with entering unexpected and terminate need to be adjusted. Issue raised by 96-0186R1/N1004R1.)

Move we amend the working paper as follows:

Current wording in 15.1p6:

> An exception is considered finished when the corresponding catch clause exits.

Revised wording:

> An exception is considered finished when the corresponding catch clause exits or when unexpected() exits after being entered due to a throw.

Current wording in 15.5.1p1:

> — when a exception handling mechanism, after completing evaluation of the object to be thrown but before completing the initialization of the exception-declaration in the matching handler, calls a user function that exits via an uncaught exception,

Revised wording:

> - when an exception handling mechanism, after completing evaluation of the object to be thrown but before either completing the initialization of the exception-declaration in the matching handler or entering unexpected() due to the throw, calls a user function that exits via an uncaught exception,

Current wording in 18.6.3.3p2 for terminate():

> Effects: Calls the terminate_handler function in effect immediately after evaluating the throw-expression (18.6.3.1).

Revised wording:

> Effects: Calls the terminate_handler function in effect immediately after evaluating the throw-expression (18.6.3.1).
> Effects: If terminate() is called by the implementation, uncaught_exception() returns true when called at any point after entry to terminate().

Current wording in 18.6.4 for uncaught_exception():

> Returns: true after completing evaluation of a throw-expression until completing initialization of the exception-declaration in the matching

handler (15.5.3). This includes stack unwinding (15.2).

Revised wording:

Returns: true after completing evaluation of a throw-expression until either completing initialization of the exception-declaration in the matching handler (15.5.3) or entering unexpected() due to the throw; or after entering terminate() for any reason other than an explicit call to terminate(). [Note: this includes stack unwinding (15.2).]

===

2. Correct the semantics of "dynamic_cast".

(Rationale: it was the intent that access checking in dyanamic_cast mimic the checking which would be done by a single static cast (for direct downcasts) or a pair of static casts to the most derived type and then to the result type (for cross-casts). The current wording does not do this, and also has an unintended bug allowing unsafe direct downcasts in some situations. Core issue 549.)

Move we amend the working paper as follows:

Current wording in 5.2.7p8:

The run-time check logically executes like this: If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class sub-object of a T object, and if only one object of type T is derived from the sub-object pointed (referred) to by v, the result is a pointer (an lvalue referring) to that T object. Otherwise, if the type of the most derived object has an unambiguous public base class of type T, the result is a pointer (reference) to the T sub-object of the most derived object.

Revised wording:

The run-time check logically executes like this:

- If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class sub-object of a T object, and if only one object of type T is derived from the sub-object pointed (referred) to by v, the result is a pointer (an lvalue referring) to that T object.

- Otherwise, if v points (refers) to a public base class sub-object of the most derived object, and the type of the most derived object has an unambiguous base class of type T, the result is a pointer (reference) to the T sub-object of the most derived object.

Current wording in 5.2.7p10:

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B {};
void g()
{
    D d;
    B* bp = (B*)&d;  // cast needed to break protection
    A* ap = &d;      // public derivation, no cast needed
    D& dr = dynamic_cast<D&>(*bp);  // succeeds
    ap = dynamic_cast<A*>(bp);      // succeeds
    bp = dynamic_cast<B*>(ap);      // fails
    ap = dynamic_cast<A*>(&dr);     // succeeds
    bp = dynamic_cast<B*>(&dr);     // fails
}
```

Revised wording:

```
class A { virtual void f(); };
```

```
class B { virtual void g(); };
class D : public virtual A, private B {};
void g()
{
    D d;
    B* bp = (B*)&d;  // cast needed to break protection
    A* ap = &d;      // public derivation, no cast needed
    D& dr = dynamic_cast<D&>(*bp);  // fails
    ap = dynamic_cast<A*>(bp);      // fails
    bp = dynamic_cast<B*>(ap);      // fails
    ap = dynamic_cast<A*>(&dr);     // succeeds
    bp = dynamic_cast<B*>(&dr);     // fails
}
```

==

3. Clarify the semantics of throwing an exception object of array or function type, as follows:

(Rationale: initializing the handler in an exception-declaration is intended to parallel initializing the parameter in a function call. But the handler type is not known at the point of the throw, and there is no way to initialize the exception temporary if the thrown expression has array or function type. So the array-to-pointer and function-to-pointer conversions should be done at the throw and the corresponding adjustments should be made to the handler type. Core issue 678.)

Move we amend the working paper as follows:

Current wording in 15.1p3:

> A throw-expression initializes a temporary object of the static type of the operand of throw, ignoring the top-level cv-qualifiers of the operand's type, and uses that temporary to initialize the appropriately-typed variable named in the handler.

Revised wording:

> A throw-expression initializes a temporary object, the type of which is determined by removing any top-level cv-qualifiers from the static type of the operand of throw and adjusting the type from "array of T" or "function returning T" to "pointer to T" or "pointer to function returning T," respectively. The temporary is used to initialize the appropriately-typed variable named in the handler.

New paragraph between 15.3p1 and 15.3p2:

> A handler of type "array of T" or "function returning pointer to T" is adjusted to be type "pointer to T" or "pointer to function returning T", respectively.

==

4. Clarify when class templates are instantiated during a "trial parse".

(Rationale: template-ids used as qualifiers must be instantiated to determine whether the qualified name is a type. So the working paper note to the effect that instantiation is not done in a trial parse is incorrect. The proposed resolution makes any program for which it matters ill-formed. Core issue 671.)

Move we amend the working paper as follows:

Current wording in 6.8p3:

> The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are

type-ids or not, is not used in or changed by the disambiguation. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. [Note: because the disambiguation is purely syntactic, template instantiation does not take place during the disambiguation step.]

Revised wording:

> The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are type-ids or not, is not used in or changed by the disambiguation. Class templates appearing in qualifiers are instantiated as necessary to determine if the qualified name is a type-id. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [Note: this can occur only when the name is declared earlier in the declaration.]

==

5. Editorial change to restore intended interaction of default arguments and using-declarations.

(Rationale: the working paper was changed in a way not intended by a motion about default arguments. The original intent is restored.)

Current wording n 8.3.6p9:

> When a declaration of a function is introduced by way of a using declaration (7.3.3), any default argument information associated with the declaration is imported as well. If the function is redeclared thereafter in the namespace with additional default arguments, the imported declaration is not affected.

Revised wording:

> When a declaration of a function is introduced by way of a using declaration (7.3.3), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also made known through the using declaration.

==

6. Clarify when overload resolution causes class template instantiation.

(Rationale: the working paper implies that every possible conversion of an argument to a potential parameter must be considered. This can result in a massive number of class template instantiations, most of which can usually be avoided by a careful implementation of the overload resolution algorithm (because a function is known to be nonviable or not the best match through information about other parameters. Core issue 676.)

Move we amend the working paper as follows:

Remove the editorial box #14 in 14.7.1p3, and add a new paragraph in its place:

> If the overload resolution process can determine the correct function to call without instantiating a class template definition, it is unspecified whether that instantiation actually takes place. [Example:

```
template <class T> struct S {
        operator int();
};
```

```
void f(int);
void f(S<int>&);
void f(S<float>); // instantiation of S<float> allowed
                  // but not required

void g(S<int>& sr) {
       f(sr);     // instantiation of S<int> allowed
                  // but not required
};
```

– end example]

==
.

7. Clarify the set of associated names used for argument-dependent name lookup.

(Rationale: the working paper does not address how any non-type arguments and template template arguments in a template-id affect the set of associated names. Core issues 686 and 703.)

Move we amend the working paper as follows:

Add the following footnote to 3.4.2p2, after "… be considered;":

> The set of namespaces is determined entirely by the types of the arguments. Typedef names used to specify the types do not contribute to this set.

Current wording in 3.4.2p2:

> — If T is a template-id, its associated namespaces are the namespace of the template and the namespaces associated with the type of template arguments.

Revised wording:

> – If T is a template-id, its associated namespaces are the namespaces of the template, the namespaces associated with the types of the template arguments provided for template type parameters (excluding template template parameters), and the namespaces of any template template arguments.

==

8. Editorial change to clarify a description of exception type matching.

(Rationale: there were two ways to interpret the text. It was clear to Core-III which one was intended.)

Current wording in 15.3/2:

> A handler is a match for a throw-expression with an object of type E if
> ...
> — the handler is of type cv1 T* cv2 and E is a pointer type that can be converted to the type of the handler by a standard pointer conversion (4.10) not involving conversions to pointers to private or protected or ambiguous classes, or a qualification-conversion (4.4), or a combination of these two.

Revised wording:

> A handler is a match for a throw-expression with an object of type E if
> ...
> — the handler is of type cv1 T* cv2 and E is a pointer type that can be converted to the type of the handler by a combination of

- a standard pointer conversion (4.10) not involving conversions to pointers to private or protected or ambiguous classes

- a qualification conversion

==

9. Clarify that member templates do not suppress implicit copy/assign functions.

(Rationale: The automatic declaration of implicit copy constructors and copy assignment operators should not be affected by the possibility of a member template specialization meeting the requirements for a copy constructor or copy assignment operator. Issue 8.3 in 96-0158/N0976.)

Move we amend the working paper as follows:

Current wording in 12.8p2:

> A constructor for class X is a copy constructor if its first parameter is of type X&, const X&, volatile X& or const volatile X&, and either there are no other parameters or else all other parameters have default arguments (8.3.6).

Revised wording:

> A non-template constructor for class X is a copy constructor if its first parameter is of type X&, const X&, volatile X& or const volatile X&, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [Footnote: Because a template constructor is never a copy constructor, the presence of such a template does not suppress the implicit declaration of a copy constructor. Template constructors participate in overload resolution with other constructors, including copy constructors, and a template constructor may be used to copy an object if it provides a better match than other constructors.]

Current wording in 12.8p9:

> A user-declared copy assignment operator X::operator= is a non-static member function of class X with exactly one parameter of type X, X&, const X&, volatile X& or const volatile X&.

Revised wording:

> A user-declared copy assignment operator X::operator= is a non-static non-template member function of class X with exactly one parameter of type X, X&, const X&, volatile X& or const volatile X&. [Footnote: Because a template assignment operator is never a copy assignment operator, the presence of such a template does not suppress the implicit declaration of a copy assignment operator. Template assignment operators participate in overload resolution with other assignment operators, including copy assignment operators, and a template assignment operator may be used to copy an object if it provides a better match than other assignment operators.]

==

10. Assume a dependent qualified name is a type in some additional contexts.

(Rationale: There are to two contexts where a qualified name is assumed to be a type, and the typename keyword is neither necessary nor permitted to be applied to the name. The two contexts are qualifiers (e.g. B in A::B::C) and elaborated names (e.g. B in struct A::B). But there are also two contexts where a qualified name must be a type name, and the typename keyword is not permitted, yet the name is not assumed to be a type name. (That is, typename is both required and forbidden.) These contexts are base class specifiers and member/base initializers. These two cases should be made to behave like the first two, i.e. typename is assumed. Core issue 736.)

Move we amend the working paper as follows:

> Add a new paragraph between 14.6p3 ad 14.6p4:
>
>> The keyword typename is not permitted in a base-specifier or in a mem-initializer. In these contexts a qualified-name that depends on a template parameter is implicitly assumed to be a type name.

==

11. Editorial change to clarify that “friend class T” is not permitted.

(Rationale: This is a frequently asked question. Core issue 738.)

> Current wording in 7.1.5.3p5:
>
>> If the identifier resolves to a typedef-name or a template type-parameter, the elaborated-type-specifier is ill-formed.
>
> Revised wording:
>
>> If the identifier resolves to a typedef-name or a template type-parameter, the elaborated-type-specifier is ill-formed. [Note: This implies that, within a class template with a template type-parameter T, the declaration "friend class T;" is ill-formed.]

==

12. Clarify that declarations of references to functions and pointers to members may have exception-specifications.

(Rationale: It was an oversight that these cases were left out. Core issue 740.)

Move we amend the working paper as follows:

> Current wording in 15.4p1:
>
>> An exception-specification shall appear only on a function declarator in a function or pointer declaration or definition.
>
> Revised wording:
>
>> An exception-specification shall appear only on a function declarator in a function, pointer, reference, or pointer-to-member declaration or definition.”

==

13. Clarify that when a pointer to member is dereferenced, the class types associated with both the object and the pointer to member must be complete.

(Rationale: If the classes are different, they must be complete to perform required conversions. Even if the classes are the same, allowing the class to be incomplete would provide little additional functionality and might overly constrain some implementations of pointers to members. Core issue 644.)

Move we amend the working paper as follows:

> Current wording in 5.5p2 and 5.5p3:
>
>> The binary operator .* binds its second operand, which shall be of type “pointer to member of T” to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.
>>
>> The binary operator ->* binds its second operand, which shall be of

type “pointer to member of T” to its first operand, which shall be of type “pointer to T” or “pointer to a class of which T is an unambiguous and accessible base class.” The result is an object or a function of the type specified by the second operand.

Revised wording:

> The binary operator .* binds its second operand, which shall be of type “pointer to member of T” (where T is a completely defined class type) to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.
>
> The binary operator ->* binds its second operand, which shall be of type “pointer to member of T” (where T is a completely defined class type) to its first operand, which shall be of type “pointer to T” or “pointer to a class of which T is an unambiguous and accessible base class.” The result is an object or a function of the type specified by the second operand.

==

14. Editorial change to clarify that types used in exception-specifications must be complete.

(Rationale: We decided that types used in exception-specifications must be complete, but the working paper is not consistent on this point.)

Current wording in 15.4p1:

> A type denoted in an exception-specification shall not denote an incomplete type.

Current wording in 15.4p7:

> An exception-specification can include identifiers that represent incomplete types.

Revised wording in 15.4p7:

> An exception-specification shall not include identifiers that represent incomplete types.

==

15. Editorial change to add "export" to the list of keywords.

(Rationale: the keyword list is not correct.)

Current wording in 2.11p1 table 3:

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

Revised wording:

(same table with "export" added)

==

16. Correct problems in the template grammar.

(Rationale: Some nonterminals defined in the working paper are not connected to the rest of the grammar, i.e. the grammar does not say where they may be used. Also since the old use of "specialization" now means "explicit specialization", the grammar should be updated to account for this. Also, the production for elaborated-type-specifier is split into two places, and so the incomplete first production is the only one which appears in the grammar (due to the typesetting conventions. And the production for full-template-argument-list is used in only one place and can be eliminated.)

Move we amend the working paper as follows:

Original wording in 7p1:

```
declaration:
   block-declaration
   function-definition
   template-declaration
   linkage-specification
   namespace-definition
```

Revised wording:

```
declaration:
   block-declaration
   function-definition
   template-declaration
   explicit-instantiation
   explicit-specialization
   linkage-specification
   namespace-definition
```

Original wording in 7.1.5.3p1:

```
elaborated-type-specifier:
   class-key ::opt nested-name-specifier opt identifier
   enum ::opt nested-name-specifier opt identifier
```

Revised wording:

```
elaborated-type-specifier:
   class-key ::opt nested-name-specifier opt identifier
   enum ::opt nested-name-specifier opt identifier
   typename ::opt nested-name-specifier identifier
   typename ::opt nested-name-specifier identifier < template-argument-list >
```

Original wording in 14.6p2:

A qualified-name that refers to a type and that depends on a template-parameter (14.6.2) shall be prefixed by the keyword typename to indicate that the qualified-name denotes a type.

```
        elaborated-type-specifier:
           . . .
           typename ::opt nested-name-specifier identifier full-template-
argument-list opt
           . . .

        full-template-argument-list:
           < template-argument-list >
```

Revised wording:

A qualified-name that refers to a type and that depends on a template-parameter (14.6.2) shall be prefixed by the keyword typename to indicate that the qualified-name denotes a type, forming an elaborated-type-specifier (_dcl.type.elab_).

```
elaborated-type-specifier:
   . . .
   typename ::opt nested-name-specifier identifier
   typename ::opt nested-name-specifier
                  identifier < template-argument-list >
   . . .
```

Original wording in 14.7.3p1:

```
specialization:
   template < > declaration
```

Revised wording:

```
explicit-specialization:
   template < > declaration
```

==