Doc. No.: X3J16/96-0186R1
WG21/N1004R1
Date: Nov 4, 1996
Project: Programming Language C++
Reply to: Servatius Brandt, SNI
<Servatius.Brandt@mch.sni.de>

Proposal

to clarify the semantics of uncaught_exception() when called from unexpected() or terminate()

CONTENTS

1. Problems with the current definition

The current definition of uncaught_exception() in 18.6.4 is:

> bool 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).
>
> Notes: When uncaught_exception() is true, throwing an exception can result in a call of terminate (15.5.1).

There are three problems with this definition:

a) An exception that results in unexpected() or terminate() being called is considered caught by unexpected() or terminate(), resp. (see 15.1 /6). However, uncaught_exception() returns true when called from the unexpected or terminate handler although (in case of unnested exceptions) there is no uncaught exception at all:

```
#include <exception>
#include <cassert>
#include <cstdlib>

void f() throw(int)
{
    throw "x";
}

void uh()
{
    assert(uncaught_exception() == true);
    exit(1);
```

```
}

int main() {
    set_unexpected(uh);
    f();
}
```

That is, the functionality and the name of uncaught_exception() disagree.

b) The definition of uncaught_exception() says when it is true, but not when it is false. The reason is that exceptions may be nested, so that if one exception gets out of the range for that uncaught_exception() returns true, there may be another exception in that range. (An implementation can use a counter of how much exceptions are in this range.) Because this range is not ended when an exception is caught by unexpected(), uncaught_exception() remains true throughout the rest of the program:

```
  #include <exception>
  #include <cassert>
  #include <cstdlib>

void f() throw(int)
{
    throw "x";
}

void uh()
{
    throw 1;
}

int main() {
    set_unexpected(uh);
    try { f(); } catch (int) { }

    // The "x" exception was not caught by a handler:
    assert(uncaught_exception() == true);

    try { throw; } catch (const char *) { }
}
```

Moreover, as an exception is "considered finished when the corresponding catch clause exits" (15.1 /6), the exception last recently caught and not finished is "x", so that the program exits normally (after catching the rethrown "x").

c) The intention of uncaught_exception() is to detect situations where the throw of a new exception might cause abandoning of a throw for that uncaught_exception() returns true. Any such throw that is still active when terminate() is called, however, will never be abandoned when throwing a new exception because it is not allowed to exit the terminate handler by throwing an exception. That is, the uncaught_exception() return value is currently meaningless when called from terminate().

2. Solution for uncaught_exception() in unexpected()

If an exception violates an exception specification, unexpected() catches the originally thrown exception and replaces it by a new one (except the unexpected handler terminates the program). uncaught_exception(), when called after entering unexpected() and before throwing any new exception, should therefore return the same value as it would have returned when called
just before throwing the original exception. In other words, uncaught_exception() should return true after evaluation of the original throw-expression until unexpected() is entered and after evaluation of the throw-expression that exits unexpected() until completing initialization of the exception-declaration in the matching handler. The value in between is false normally but can be true when the originally thrown exception was nested or when an exception is locally thrown and caught by a function called from unexpected().

When unexpected() is not called due to a throw, but explicitly called by the user, the value of uncaught_exception() should not be affected.

[For an implementation using a counter of uncaught exceptions, this means to decrement it when calling unexpected() due to a throw but to leave it untouched when unexpected() is explicitly called by the user.]

The exception caught by unexpected() should be considered finished when unexpected exits by throwing an exception. Otherwise, the exception caught by unexpected() could be rethrown later on (when unexpected() is no longer active).

The changes proposed above solve problems a) and b).

3. Solution for uncaught_exception() in terminate()

The terminate handler is not allowed to exit by throwing an exception. If it does, the behavior is undefined. uncaught_exception() cannot be used to protect against doing so because when called from terminate() it may return *false*: when terminate() was called because no exception could be rethrown or when it was explicitly called by the user. However, throwing
from terminate can easily be avoided by the user: by checking for a global flag set in the terminate handler.

However, as uncaught_exception() can be used to avoid terminate being called
at all, the same checks (in destructors and copy constructors) help to protect
against terminate() being called from terminate() again. uncaught_exception()
then needs only to care about throws started after terminate was called, because throws started before will not be abandoned due to new exceptions, as terminate() cannot be exited by throwing an exception. Therefore, uncaught_exception() should return false after entering terminate() until the next exception is thrown.

[For an implementation using a counter of uncaught exceptions, this means to reset it to zero in terminate(), all the same if it is called due to a throw by the implementation itself or explicitly called by the user.]

The change proposed above solves problems c).

4. Wording

A) In 15.1 /6, last sentence

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

change to

"An exception is considered finished when the corresponding catch clause
or unexpected() exits."

B) Change 15.5.1, first bullet to:

- when an 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 or entering unexpected() or terminate() due to the throw, calls a user function that exits via an uncaught exception,

Here, "or entering unexpected() or terminate() due to the throw" has been added. See also C). The footnotes need not be changed.

C) Change the definition of uncaught_exception in 18.6.4 to:

bool uncaught_exception();

Returns: true after completing evaluation of a throw-expression until completing initialization of the exception-declaration in the matching handler or entering unexpected() or terminate() due to the throw. This includes stack unwinding (15.2).

Notes: When uncaught_exception() is true, throwing an exception can cause the abandoning of the throw of a presently uncaught exception. Abandonment of such a throw results in a call of terminate (15.5.1).

Here,

- the cross reference to 15.5.3 in "Returns:" has been removed because 15.5.3 just refers to 18.6.4.

- "or entering unexpected() or terminate() due to the throw" has been added. "due to the throw" ensures that the return value of uncaught_exception() is not affected when unexpected() is explicitly called during stack unwinding.

- the definition does not say when the return value is false (because of possibly nested exceptions). For the value false in terminate() see D).

- the "Notes:" have been changed to make clear why uncaught_exception() has its name. Furthermore, separating the wording "can cause the abandoning of the throw of a presently uncaught exception" from the statement that terminate() is called in case of abandonment shows

what
problems resulting in a call of terminate() can be detected and that there still are other reasons for calling terminate() that cannot be detected by checking uncaught_exception(). The "Notes:"
intentionally
speak about the *possibility of abandoning the throw* of an uncaught exception rather than just about the *existence* of an uncaught exception, because this is the interesting information and also holds when uncaught_exception() returns false in terminate() (see D)).

D) In 18.6.3.3 (terminate()) add to "Effects":

"uncaught_exception() returns false when called from the terminate handler until the next exception is thrown (see 18.6.4).

This must be added here rather than in 18.6.3.1 (terminate_handler), because
the terminate handler could also be explicitly called by the user.