Document: N2619=08-0129
Date: 2008-03-25
Reply to: Alisdair Meredith <alisdair.meredith@codegear.com>

CONCEPTS FOR CLAUSE 18

INTRODUCTION

Library clause 18 covers the library types supplied for language support. There are two main uses of templates to consider constraining, `numeric_limits` and some exception support APIs.

This paper recommends leaving `numeric_limits` as an unconstrained template. It was provided as a user customization point and so there are many valid unconstrained specializations of this template in use today. Adding constraints at this point could cause un-necessary migration issues. Note that paper n2591 suggests an alternate API for `numeric_limits` that is backwards compatible for legacy code, but extensible for future uses. This might be the appropriate way to explore constraints.

The three exception support APIs shall each be considered on their own merits.

CONSTRAINING EXCEPTION APIS

There are three function templates to consider:

template<class E> exception_ptr copy_exception(E e)

This function creates an `exception_ptr` from the passed object. The only requirement is that object is 'throwable'. The question of whether `CopyConstructible` is a requirement for 'throwability' is explored below.

The goal of this function is to provide a more efficient way to create `exception_ptr` objects than artificially throwing and catching `e` as an exception. As this requires knowledge of the compiler implementation of `exception_ptr`, can this function be implemented efficiently without further constraints? Given the minimal constraints on what is throwable, we argue that this must be possible, but will be happier once we have a sample implementation that demonstrates this.

template<class E> void throw_with_nested(E&& t)

This function is specified to cope with several special cases that might be handled by concept-based overloading. With concepts, the problem cases might instead be excluded by the requirements of the template. This would better alert users that their exception type is not suitable for the API (rather than silently failing to nest an exception at runtime) at the expense that it cannot be used quite so portably in generic code.

The direction of this paper is to assume the design goal of concepts is to better identify those problem cases, so flagging them as a requirements-violation error would be the preferred solution.

There are two sets of special cases to consider:

I/ THE TYPE E CANNOT BE DERIVED FROM.
Typically E is a scalar type such as a pointer, integer or enum, but this also covers arrays, function-types and unions.

II/ E ALREADY HAS A NESTED EXCEPTION
As this API works with the static type of `E`, rather than the dynamic, this can be easily tested with the type trait `is_base_of`. It would be a mistake to provide a separate overload taking `nested_exception` by reference, as the subsequent throw would slice the exception object.

So the fundamental requirements are that `E` is a throwable (see below) non-union class type.

It is believed that in light of n2576 the preferred syntax for argument-passing should be by-value, and the concept syntax should pick up efficient moving automatically in the case that copy-elision is not available.

template<class E> void rethrow_if_nested(const E& e)
This API tests to see if E contains a nested exception via `dynamic_cast`. Therefore the key requirement is that `E` is a polymorphic type.

This `dynamic_cast` could be avoided in many cases if we provide an additional overload taking a `nested_exception` by reference.

The remaining question is if we should add yet another overload to support non-polymorphic types in generic code – they would be supported by this API today, although the function will return without effect. As per the decision for `throw_with_nested`, we believe that is an artificial over-genericity that the concepts feature now allows us to diagnose.

WHAT IS A 'THROWABLE' TYPE?
The big outstanding question is what makes a throwable type?

The key requirements come from 15.1p3

> A temporary will be made by copying the object
>
> "When the thrown object is a class object, the copy constructor and the destructor shall be accessible, even if the copy operation is elided."
>
> "The type of the throw-expression shall not be an incomplete type, or a pointer to an incomplete type other than (possibly cv-qualified) void."

The first two requirements are covered by the `CopyConstructible` requirement, as is the complete-type requirement. The question of whether we need to constrain against pointers-to-incomplete-types comes down to what the compiler will do when encountering

```
throw T;
```

inside a constrained function-template body. This is an interaction of the core language with a constrained type, and it would require a language-support concept in order to type-check the template if that was desired. This does not appear to be the approach taken by the core language and library concepts papers, so throughout this paper `CopyConstructible` is used as a synonym for `Throwable`, and additional requirements are placed on functions as necessary. These might be enforced with a `static_assert` , or simply allowing the instantiation to fail.

Proposed Wording

Update 18.7p1 [support.except]

```
template<~~class~~CopyConstructible E> exception_ptr copy_exception(E e);
template <~~class~~CopyConstructible ~~T~~E> requires Class<E> void
throw_with_nested(T&&E e ~~t~~); // [[noreturn]]
template <~~class~~CopyConstructible E> void rethrow_if_nested(const E& e);
void rethrow_if_nested(const nested_exception & e);
```

Update 18.7.5p11 [propagation]

```
template<~~class~~CopyConstructible E> exception_ptr copy_exception(E e);
```

Update 18.7.6 [except.nested]

```
namespace std {
  …
  template <~~class~~CopyConstructible ~~T~~E> requires Class<E>
    void throw_with_nested(T&&E e ~~t~~); // [[noreturn]]
  template <~~class~~CopyConstructible E> void rethrow_if_nested(const E& e);
  void rethrow_if_nested(const nested_exception & e);
}

  template <~~class~~CopyConstructible ~~T~~E> requires Class<E>
    void throw_with_nested(T&&E e ~~t~~); // [[noreturn]]
```

~~6 *Requires:* T shall be `CopyConstructible`~~

7 *Throws:* ~~If T is a non-union class type not derived from nested_exception, a~~An exception of unspecified type that is publicly derived from both ~~T~~**E** and `nested_exception`~~, otherwise t~~.

```
  template <~~class~~CopyConstructible E> void rethrow_if_nested(const E& e);
  void rethrow_if_nested(const nested_exception & e);
```

***Requires:* E is a polymorphic type.**

8 *Effects:* Calls `e.rethrow_nested()` ~~only~~ if **the dynamic type of** `e` **(or `*e` if `E` is a pointer type)** is publicly **and unambiguously** derived from `nested_exception`.

[Draughting note: the goal of this API is to test for derivation from nested_exception with dynamic_cast, and then rethrow any nested exception that may be present. If 'dynamically derived' does not say that, we need a phrase that does. Also, would be preferable to add a requires clause to say Polymorphic<E>, but there is no such core concept at this point.]

[Draughting note 2: E does not actually need to be CopyConstructible as no copies are made, the only real requirement is E is a polymorphic class type, or a pointer to a polymorphic class type.]

ACKNOWLEDGEMENTS

Doug Gregor and Jens Maurer gave valuable feedback on early versions of this paper.