THE STATIC CHECKING OF C++ EXCEPTION SPECIFICATIONS

Andrew Peter Marlow

APM Software Ltd
44 Stanhope Road, North Finchley
London N12 9DT, England
apm35@student.open.ac.uk

ABSTRACT

This document describes an extension to C++ in the area of exception handling that is designed to facilitate static checking of exceptions. It discusses the problems with such static checking and how they may be overcome.

1. INTRODUCTION

Koenig and Stroustrup wrote a paper on the C++ exception handling mechanism (Ref [1]) which discusses many of the reasons behind the decisions taken in the way C++ exceptions work. This includes the reasons for preferring run time checking to compile time checking in the area of exception specification consistency. This document proposes an extension to C++ that allows static checking of exception specifications. It deals with the points raised in Ref[1] and discusses the common objections to static checking that have been raised on USENET.

The extension is an alternative way of specifying the exceptions that a method throws (directly or indirectly). This is done by using the new keyword *onlythrow* rather than throw in the exception specification. In just the same way that throw() indicates that no exception can be thrown, onlythrow() indicates that no exception can be thrown. In general the same rules that apply to the existing exception specification also apply to onlythrow. These are dealt with in Ref [5], section 15.4.

This extension switches the emphasis from run-time checking to compile time checking. Use of onlythrow rather than throw changes the nature of the consistency checks in two ways: firstly, exception propagation is single-level rather than multi-level, and secondly, inconsistent exception handling is treated as an error rather than a warning. This extension does not alter the behaviour or meaning of any prior code and its use is optional.

Consider an example: let us write the function f() that can only throw the exceptions e1 or e2. Using current exception specifications this would be written as:

```
void f() throw (e1, e2)
{
    ...do something...
}
```

However, the consistency of the the throw(e1, e2) clause with the associated code is not checked at compile time. Instead it is as if code were generated that makes the throwing of anything else an unexpected error at runtime, thus:

```
void f()
{
    try
    {
        ...do something...
    }
    catch (e1) { throw; /* rethrow */ }
    catch (e2) { throw; /* rethrow */ }
    catch (...) { unexpected(); }
}
```

If f() was changed to use onlythrow as shown below:

```
void f() onlythrow (e1, e2)
{
    ...do something...
}
```

then the developer would know that the body of f()had been checked by the compiler and that f() could never call unexpected().

2. WHAT CHECKS ARE PERFORMED

When a function, f(), has an onlythrow list the compiler generally makes the same checks as for the current exception specifications mechanism. These are given in Ref [5], section 15.4. However there are some differences.

Clause 8 says that ”whenever an exception is thrown and the serch for a handler encounters the outermost block of a function with an exception specification, the function unexpected is called if the exception specification does not allow the exception” and it gives an example. This is a case where using onlythrow would give rise to a compilation error.

Clause 10 specifically allows what must be rejected when the keyword onlythrow is used (An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow).

Clause 11 says that a function with an empty exception specification, throw(), does not allow any exceptions. Strictly speaking this is not accurate. Such functions are allowed to compile, but if they encounter any exception at runtime, unexpected() will be called. Using onlythrow() instead causes the compiler to check that no exceptions can be thrown. If any can then the code will not compile. Thus the developer has the assurance that a function with the new empty exception specification, onlythrow(), does not allow any exceptions at compile time or runtime.

Clause 13 describes how classes may acquire implicit constructor/destructor declarations that contain exception specifications. When onlythrow is used instead of throw the compiler shall

not generate the implicit constructor/destructor code. This is because the developer must always actively select the static checking behaviour of the new exception specification mechansim.

Functions that use onlythrow exception specifications cannot call functions that use throw for their exception specifications. This is because such functions may violate their exception specifications and this is only detected at runtime. This aspect of onlythrow is covered in more detail later.

The exception specification consistency checks are also made whenever template code is instantiated that employs the onlythrow directive. It is recommended that template classes do not use onlythrow since template classes usually use their template argument in some way and the template argument might throw exceptions that the template class does not known about. With the standard exception specification this would be accepted by the compiler but could cause a fatal runtime error. This is warned about in Meyers II, item 14 (Ref [3]). Using onlythrow would cause the code to fail to compile which is better but still undesirable because such code would be sensitive to the kinds of template argument that might be supplied and would force the template argument class to change so that the methods invoked by the template code did not violate the onlythrow list. The problems with templates and the stricter checking that onlythrow affords were frequently raised on USENET as a reason for not having stricter checking. However, the problems exist even with the current exception specification facility.

3. THE PROPAGATION OF EXCEPTIONS

In D&E (ref [2]), section 16.8, Stroustrup argues that to allow an exception to be propagated from a function to its immediate caller only was not an option for C++. Among the reasons given is: It is not a good idea to try to make every function a fire-wall. The best error-handling strategies are those in which only designated major interfaces are concerned with non-local error-handling issues. This argument is a precursor to a statement he makes in the next section: By allowing multi-level propagation of exceptions, C++ loses one aspect of static checking. One cannot simply look at a function to determine which exceptions it may throw. The extension proposed in this document is a proposal for a form of exception handling that is single-level propagation only. It must therefore deal with the objections raised in D&E.

This proposal conceeds that it is not feasible to modify all existing C++ code to propagate or handle exceptions. That is why it is beneficial to have this proposal as an optional extra, rather than a replacement for the way exception-specifications work at present.

It is a matter of opinion whether it is a good idea to make every function a fire-wall. Eiffel, for example, takes the view that it is. C++ takes the view that it is not. There are many developers that would like to see a stronger form of contractual obligations and responsiblities in C++ methods, as is the case in Eiffel with its Design by Contract (DbC) form of pre and post condition statements. This extension is not intended to offer DbC despite a degree of convergence in this direction, neither is it is an attempt to make functions into fire-walls. It is simply an attempt to offer an optional form of stronger static checking for those developers that want it, as discussed in D&E section 16.9. This does seem to be a goal of exception handling in C++ judging from what is said in 16.9, particuarly when it says ”Ideally, exception specifications would be checked at compile time, but that would require that every function cooperates in the scheme, and that isn't feasible”. When a developer uses onlythrow exception specifications it does not require that every function cooperates with the scheme. Only functions that have an onlythrow exception specification are subject to the stricter static checking. Functions that do not have an onlythrow exception specification are not affected even if they call functions that do have one.

A C++ function may be called by a C function or indeed a function written in some other language. This is a case where multi-level propagation is needed. It is acknowledged that in such cases the onlythrow keyword should not be used but neither should the existing exception specification syntax. Functions that need to propagate any exceptions that they throw to an outside world that cannot immediately deal with them certainly exist and C++ offers a way to deal with this: such functions simply don't employ exception specifications.

This proposal argues that exception handling is improved by the ability to declare functions that only propagate to the immediate caller. At the same time it acknowledges that there are valid reasons to prefer multi-level propagation in some cases. This is why the facility is introduced via a new keyword. It provides an additional facility rather than a replacement.

4. STATIC CHECKING

It is a design philosophy in C++ that compile time checking is preferred to run-time checking. This extension helps to realise this goal in the area of exception handling. However, this stricter checking comes at a price: it forces the developer to treat the function prototype including any exception specification as a binding contract.

This rigour may be too much for some developers since a change to the exception specification of a low level routine will affect the callers at the higher level, forcing the caller to deal with the change. This is also true of any changes to the return type or parameter list but developers seem to accept this consequence more readily. It is interesting to note that when prototypes were first introduced to ANSI C this rigour was initially resented by some who sought to trick the compiler by declaring all routines to have a parameter list of (...). Preliminary discussions of this proposal on USENET have caused similar sentiments to be expressed to those that sought to subvert the stronger checking that ANSI C prototypes brought to the K&R C community. Reactions such as these to a strong exception specification checking mechanism is anticipated in D&E section 16.9 where it describes the situation of a change in an exception specification that ripples up. It quotes from Ref [1], saying ”Such problems would cause people to avoid using the exception specification mechanism or else subvert it”.

This proposal does acknowledge that the ripple effect problems described in D&E 16.9 will cause some people to avoid using the exception specification mechanism or else subvert it. However, this is no reason not to propose it. The ANSI committe went ahead with the use of function prototypes in C despite the resentment by some developers and despite the tricks they resorted to in order to avoid the rigour such prototypes impose. The language became better for it and the rigour is now accepted as good practise by all. It is hoped that the proposal will be accepted and that in the fullness of time the extra rigour afforded by the use of the new exception specification syntax will be viewed as part of a functions contract, in just the same way that the rest of the prototype is. To ease the transition, use of the facility is optional. Even now, ANSI compliant C compilers will allow functions to be called for which there is no prototype (although they may warn about the implicit

declaration).

5. SUBVERSION

In Ref [1], section 8.1, an example is given of how programmers might subvert a stronger form of static checking in order to insulate themselves from any changes to routines that they call (changes to the exception specification). However, the example fails to point out that when g() calls f() and is affected by changes to f()'s exception specification, it is only affected if g() itself has an exception specification. Just because a routine comes with an exception specification does not mean that the caller has to have one. The static checks that this document describes only apply to those functions that use onlythrow for their exception specifications. Functions that use the existing form of throw list or that do not have a throw list at all do not need the checks that onlythrow uses.

The example also says "To allow such checking every function must be decorated with the exceptions that it might throw". This is not true. If it were then this proposal would indeed be impractical. Consider two routines, let us call them A and B: A is a routine that employs onlythrow and it calls B which does not have an exception specification. A must treat B as a routine that might throw anything and calls to it must be in a try block that has an associated catchall, catch(...). If A calls many such routines then it might prove tiresome to protect each one with:

```
try { routine... }
catch(...)
    { map exception to one in my only-list}
```

In this case it might be easier for A to have one try block with the `catch(...)` at the end.

Treating B as a routine that might throw anything is, of course, what happens with C++ by default. In Ref [1], section 8.1 it says that this puts an intolerable burden on the programmer who wishes to write a function that is restricted to a particular set of exceptions. It points out that the programmer would have to guard against every call to every unrestricted function. This is true but by the use of onlythrow any such calls that the programmer forgot about would be flagged by the compiler. Functions that wish to restrict themselves to a particular set of exceptions they can throw do not need to catch every kind of exception that might be thrown by the routines that they call. This is because such exceptions can be caught via a base class.

The fact that this proposal only places the burden of stronger exception checking on developers that want to use it can be used to refute some of the examples that are sometimes used to show how a developer might be persauded to subvert the system. Consider the developer routine absSqrt which returns the positive square root of the absolute value of the argument and guarantees not to throw any exceptions. Its signature is:

```
double absSqrt(double x) onlythrow();
```

This routine uses sqrt to do its work but imaginee that the signature for sqrt is:

```
double sqrt(double x) throw(IllegalArgument)
```

The developer wants to implement absSqrt as follows:

```
double absSqrt(double x) onlythrow()
{
    return x < 0 ? sqrt(x) : sqrt(-x);
}
```

but the compiler flags an error because `IllegalArgument` is not being handled. The developer changes the above code to:

```
double absSqrt(double x) onlythrow()
{
    try {
    return x < 0 ? sqrt(x) : sqrt(-x);
    } catch (IllegalArgument&) {};
}
```

and the code now compiles. The developer is annoyed (so it is argued) that this had to be done because he knows that the catch can never be executed but its presence makes the code longer and therefore more complex and adds a runtime cost.

It is suprising how often this objection is raised when it can be refuted by simply saying that those developers that have taken the trouble to add onlythrow() to the signature for absSqrt have done so because they know they must write code that guarantees no exceptions will be thrown. They appreciate that they cannot add the promise without adding the code that is needed to enforce it. They also know that this code comes with a runtime cost. This should make them think twice about adding such promises to performance-critical routines. Similar peformance penalties already apply to the existing mechanism.

6. WHEN NO EXCEPTIONS ARE THROWN

The onlythrow directive can be used with an empty exception list, i.e `onlythrow()`. This has the effect of saying that the function is guaranteed not to throw any exceptions and that this is enforced at compile time (unlike throw() where it is checked at runtime). This is not to be confused with use of nothrow(), which is concerned with new (it makes it return a null pointer in preference to throwing an exception).

7. PROPAGATING EXCEPTIONS FROM DESTRUCTORS

The standard (Ref [5]) does not prohibit one from doing this but points out that if this is done whilst an exception is active, terminate() will be called. Coding standards and books and articles on best practise point this out and recommend that exceptions are prevented from leaving destructors. For further examples on this see Ref [3] and Ref [4] where it is recommended that a developer always adds throw() to any destructor that they write.

Destructors from the standard C++ library tend to have throw() as part of their signature. This is not a promise that these destructors will not allow such propagation. It merely says that if they do then a runtime error will occur: unexpected() will be called. If these destructors employed onlythrow() instead, then users of these classes could be confident that the destructor would never allow an exception to be propagated outside.

The ANSI committe are currently deliberating a number of open issues. One of these, raised by Herb Sutter in March 2001, is to do with the dangers of propagating exceptions from a destructor. It reads as follows: Destructors that throw can easily cause programs to terminate, with no possible defence. Example: Given

```
struct XY { X x; Y y; };
```

Assume that X:: X() is the only destructor in the entire program that can throw. Assume further that Y construction is the only other operation in the whole program that can throw. Then XY cannot be used safely, in any context whatsoever, period - even simply declaring an XY object can crash the program:

```
XY xy; // construction attempt might
       // terminate the program:
       //   1. construct x -- succeeds
       //   2. construct y -- fails,
       //                throws exception
       //   3. clean up by destroying x --
       //      fails, throws exception,
       //      but an exception is already
       //      active, so call
       //      std::terminate() (oops)
       // there is no defense
```

So it is highly dangerous to have even one destructor that could throw.

In the example it is the compiler-generated constructor that would call terminate(). Unfortunately it is not always possible to code an explicit constructor for XY that guarantees not to throw any exceptions, even when qualifying the constructor declaration with onlythrow(). Such a constructor would fail to compile here since the compiler would spot that the throw in the X destructor prevents the XY construction from being exception-free. However, the constructor cannot always trap and deal with such exceptions. If default constructors are invoked or constructors from initialisation lists then there is no syntax available to surround them in a try block and provide catch handlers. Hence the only solution is to declare the destructor using onlythrow() and alter the destructor code so that it no longer throws an exception. The developer might not be able to do this. For example, X may come from a third-party library and so cannot be modified. There is no easy solution to this except to say that with the introduction of onlythrow(), vendors of third-party libraries have the opportunity to create class libraries whose class destructors are guaranteed not to throw exceptions. If they don't make use of it then problems like the above may persist.

There is another problem with the issue of propagating exceptions from destructors: developers that wish to write functions with onlythrow lists have to worry about the fact that these functions might destroy objects whose destructors do not have an exception specification. Such destructors must be treated by the compiler as if they might throw any exception. This means that when these objects go out of scope or are delete'd, the function must have a suitable catch clause, even though it is probably impossible for it to be executed. One way to approach this would be for the function to have a try block at the outermost level that has a catch(...) clause that does nothing (i.e it absorbs and throws away any exception that it catches).

8. IMPLICATIONS FOR THE STL AND STANDARD C++ LIBRARY

The implications for destructors in the standard library has already been mentioned. There are other functions in the standard library that are defined to throw exceptions. These functions do not have throw lists though. In order to know what exceptions are thrown the standard must be consulted. Even if the conventional form of throw list was used this would be no guarantee that this is all that might be thrown (although if another exception were thrown, unexpected() would be called). Perhaps that is why the standard functions do not employ throw lists. Using onlythrow() in these standard functions would guarantee that unexpected() could never be called. This would also make the signatures sensitive to changes in the standard. Adding new exceptions that might be thrown would require that they be added to the onlythrow list. This needn't affect the callers though unless the caller also has an onlythrow. In this case they will be affected and rightly so. The use of onlythrow will alert them to a new exception that they must handle in order to keep the promise they make about what exceptions they propagate.

The contractual nature of the C++ library interfaces specified in the C++ standard is much more tightly binding than functions developed by a developer as part of their project development. The developer's work is much more fluid and any standard it must conform to can be easily changed as the need arises. Thus the use of onlythrow in standard C++ routines is not very likely to be disruptive to the developer since changes to the standard are infrequent and the use of onlythrow by the standard does not mean that the developers own code has to employ onlythrow. The standard routines would benefit from using onlythrow since they benefit developers that want stricter exception consistency checks in their own code without it adversely affecting those that don't.

C++ defines that certain standard exceptions may be thrown even where no function call is apparantly being made. Examples include failure to dynamic cast and failure of new to allocate from the heap. If an onlythrow function contains code such as this then the compiler has to watch out for it in order to check that such exceptions do not propagate beyond the function unless they are mentioned in the onlythrow exception specification.

Section 5, clause 5 of the standard describes how some expressions may yield undefined results at runtime. Examples include divide by zero and integer underflow. Implementation-defined behaviour where the standard says that the results are undefined means that such expressions may, but are not required, to throw exceptions. A function that employs `onlythrow` for its exception specification is not prohibited from containing such expressions. This may seem counter-intuitive but if such a function encountered those conditions then whether an exception is thrown or not, the behaviour is not defined by the standard. Therefore these cases should be considered to be part of the preconditions for the function: the postconditions may be met if the preconditions are not met.

The class `exception`, which defines the base class for the types of objects thrown as exceptions by C++ standard library components and certain expressions, uses exception specifications on every member function. These should be changed so that the keyword onlythrow is used instead of throw.

9. HANDLING OF CONSTRUCTORS

One of the primary uses for exceptions is that they provide a way for constructors to fail in spite the fact that they do not return an error code. For programmers that wish to handle these exceptions it is useful for them to know what exceptions might be thrown. The throw list was intended to provide this but the contents of the throw list cannot be relied on. For this reason, the use of throw lists tend to be discouraged. This is ironic because they discourage the very thing that they were supposed to encourage. This is because the

developers that would want to use them tend to be developers that prefer stronger compile-time checking. Developers that prefer not to have exceptions checked by the compiler tend not to use throw lists at all. So whilst the use of exceptions are many, they tend to be of particular use in constructors and this is where onlythrow is particularly useful.

The use of onlythrow() in constructors does not affect developers that do not wish to be concerned with exceptions. Some code is exception-neutral, i.e it is not going to translate or absorb an exception. it merely allows such exceptions to propagate up to a caller that can handle it. This code simply does not use onlythrow. It doesn't matter that it is calling code that does use onlythrow.

10. TYPE CHECKING OF EXCEPTIONS

Overloading a function on its onlythrows list is not allowed. This means that variation in the contents of an onlythrows list for a function with a given signature is not allowed. A function may not be overloaded by providing a version with onlythrow and a version without onlythrow. A function may not be overloaded by providing a version with onlythrow and a version with throw.

C++ already has to cope with the fact that a developer may define a function twice with differing exception specifications, including no exception specification at all. The approach has been that the exception specification does not take part in function matching for overloaded functions. This proposal does not alter the situation. Such attempts will be caught by the linker as it attempts to link a duplicate definition of the function. Not all linkers alert the developer to this situation.

11. EXCEPTION HANDLING AND TEMPLATES

In Ref [3], item 14 is devoted to discussing the risks in using exception specifications in template code. After several examples it concludes with *'templates and exception specifications don't mix'*. This proposal says that this is even more the case when using `onlythrow`. Using `throw` may result in unexpected() being called when the developer did not want it to be, but using `onlythrow` will result in the code failing to compile which is also undesirable. Since template code may use code that comes from the template argument that the author of the template code is not even aware of, it is better for template functions to be exception-neutral. This is only a recommendation for developers though.

Developers that use STL containers should pay close attention to the recommendation: it means that functions that use STL containers are best coded without exception specifications.

12. MIXING THE USE OF THROW AND ONLYTHROW

There are two cases to consider:

1. A function uses throw for its exception specification and calls routines that employ onlythrow
2. A function uses onlythrow for its exception specification and calls routines that employ throw

The first case is simple: the compiler should treat the onlythrow directives in exactly the same way as it would throw directives.

The second case is a bit tricky. When an onlythrow function calls one with a conventional throw list, it has to worry about the fact the when it makes the call, unexpected may be called. This is even the case when destructors might be called that have an empty exception specification (like the ones in the standard C++ library tend to do). What happens when unexpected() is called cannot be known by the compiler at this point. It may well be the default behaviour but this could easily be overridden via set_unexpected() at any point. The developer may, for all the compiler knows, have used set_unexpected() to map unexpected errors to his own application-defined unexpected exception.

The simplest solution seems to be that functions that use onlythrow for their exception specifications cannot call routines that employ the throw keyword for their exception specifications. This means that compilers that implement onlythrow must ship with a standard library that uses it instead of using throw.

This puts pressure on the old style of exception specification to be deprecated. This is an undesirable side effect. However, even without the old style, multi-level propagation is still possible: functions that wish to propagate exceptions without having any handling code simply don't use exception specifications.

The inability for a function that has an onlythrow exception specification to call functions that have a throw exception specification raises two issues.

1. Third party libraries may use the old exception specification mechanism. If they do then they cannot be called by functions that want to use onlythrow.
2. Third party code generators, such as OMG IDL compilers, may generate code that cannot be called from a function that wishes to use onlythrow.

The first point is not the serious problem that it may seem to be at first glance. The lack of an ABI in C++ already forces the vendors of third-party C++ libraries to reproduce *exactly* the compilation environment of the people that they are trying to sell to. The vendor's library must already be compiled with the same compiler, at the same release, with the same compiler switches, on the same operating system at the same version with the same patches, as the customers code that wishes to use it. When compilers become available that implement onlythrow, vendors will have to make a decision as to whether or not their library supports such a compiler. If it does then the library code may have to be altered to use onlythrow in exception specifications that previously used the throw keyword.

Similar considerations apply to third-party code generators except that in the case of OMG IDL there may need to be a change to the OMG standard that defines the mapping from IDL to C++.

13. COMPARISON WITH JAVA EXCEPTION HANDLING

Ref [1], section 8.2 discusses the dangers of a compiler that produces too many warnings about exceptions that are not being caught and handled and suggests that if developers really want this sort of checking that it might be provided by some separate tool in the future. It is certainly true that developers resent being showered by spurious warnings. However, this proposal is for onlythrow to cause errors when exceptions are not being handled rather than warnings and the errors are only issued when the developer chooses to write code that makes guarantees about what will be thrown then inadvertently breaks those guarantees. Just calling code that employs onlythrow will not cause such errors to be raised: the developers own functions must employ onlythrow and not handle one or

more of the exceptions in that list. By the use of base class exceptions it is an easy matter to catch and handle groups of exceptions that map onto a single exception in the developers onlythrow list.

A potential danger with onlythrow is that once a developer has decided to write functions that use onlythrow they get showered with errors from the compiler either because they are calling routines that throw exceptions they were not aware of, or becaause they are calling routines with no exception specification and have not used catch(...) around such calls. This situation could be exacerbated by using many routines with onlythrow specifications. There are several sources of such calls:

1) The standard C++ library

onlythrow should be used in moderation in the standard C++ library. Lessons can be learnt from Java here, whose foundation classes often throw exceptions that have to be dealt with by the immediate caller. This tends to de-sensitize the developer to such exceptions and sometimes discard them (just to shut the compiler up).

2) Third-party class libraries

It is possible that the developer is forced to use a class library whose routines tend to use onlythrow in an excessive way. Imagine the inconvenience to the developer if such a library used onlythrow(IntegerOverflow) for every function that had an integer parameter and the developer wanted to call such routines from methods he was writing and which were using onlythrow lists.

3) Code written by the developer

It is up to the developer to strike the balance between code that never throws exceptions and code that pays excessive regard to exceptions, which should be exceptional, after all.

The dangers above are sometimes argued as a reason for not employing strong checking in exception specifications, particularly by Java programmers that have had a taste of the Java approach of single level propagation. During early USENET discussions this proposal was compared to the way exception specifications are done using Java. There are several ways to deal with the dangers above and some important differences between Java's approach and the approach of C++ with the onlythrow facility:

Java's stricter exception checking policy is not optional: use of onlythrow is. C++ developers that prefer the stricter approach are advised to get used to adding onlythrow() to any functions that they write. Then the compiler will tell them what exceptions they are failing to handle much as Java does.

When a developer calls a function that uses the stricter checking this does not force the developer to be strict, unlike Java. In Java if a method has a throw list then anyone who calls it has to handle it otherwise they get a compilation error. This aspect of Java was mentioned repeatedly when this proposal was discussed on USENET and was cited as a reason for rejecting the proposal. The fear was that standard C++ library routines would use onlythrow and thus might force developers to add exception handling code to their functions that was not previously required.

The danger of excessive use of onlythrow by third-party libraries certainly exists and has no easy solution. Developers will have to be patient as the developers of such libraries grow more experienced in its use. In time the use of onlythrow will probably peak and then diminish to the minimum essentials. This situation will be easier with the developer's own code which can be changed at will.

There is an important aspect to Java exception handling that this proposal does not wish to emulate: in Java static checking is ignored for exceptions that derive from runtime exception. In C++ there is a `runtime_exception` base class as part of the standard hierachy. This proposal does not intend that exceptions that inherit from runtime_exception are exempt from static checking. The meaning of onlythrow is always that the exceptions in the onlythrow list are the only ones that can be thrown and this is true no matter what they inherit from and no matter what other exceptions may exist and no matter what they inherit from.

It is also said that Java exception handling is fragile: that a change to the exception specification for a low level routine can easily ripple way up and impact a large amount of code. One has to wonder in cases like this why more use is not made of the exception hierarchy. A new exception that inherits from a base class already in the onlythrow list would have no ripple effect at all unless some code higher up was specifically interested in this new exception (in which case the handling code would have to be added anyway). One has to presume that the new exception is not related to the other exceptions. If this really is the case then perhaps it is right that there is a ripple effect. After all if something new can happen then all the potentially affected code ought to know about it. The risks of this happening as a project is in the early stages can be reduced by establishing an exception hierachy early on and strongly encouraging the use of exception inheritance.

Another point worth making is that many developers feel that Java overuses exceptions for situations that would be handled using error codes in other languages. In those situation C++ developers ought to consider error codes. This is another way of minimising the ripple effect when a low level routine changes. Error codes are often implemented using an enum but Java does not have enums. Perhaps this is why Java uses exceptions where C++ code would find the error code approach more useful.

Java exception handling provides a useful example to C++ of how static checking can be peformed and the benefits it gives. It demonstrates that it is possible to implement a compiler that makes these checks. Java exception handling is sometimes troublesome because it is compulsory for code that calls a function with an exception specification. However the same is not true for C++ since the checks are only made for functions that have an onlythrow exception specification.

14. SUMMARY

1. The keyword `onlythrow` shall be used to denote an exception specification that is subject to static checking and for a function that uses single level propagation of exceptions.
2. Functions that use `onlythrow` for their exception specification are checked to ensure that exceptions not in the exception specification cannot propagate outside the function. For any that do a compilation error shall result.
3. Functions that use `onlythrow` for their exception specification are not allowed to call functions that employ the `throw` keyword for their exception specifications. Any such attempt shall result in a compilation error.
4. A compiler that offers `onlythrow` shall use it in the STL it supplies in the form `onlythrow()` on destructors only where currently the standard has `throw()`;
5. Functions that do not have the `onlythrow` form of exception specification may call functions that do.

6. A Function that uses `onlythrow` for its exception specification may contain expressions that under certain conditions yield undefined behaviour, according to the standard.
7. A Function that uses `onlythrow` for its exception specification is **not** allowed to suspend the consistency checking for exceptions that inherit from the standard `runtime_exception` base class.

15. REFERENCES

[1] Exception Handling for C++ (revised), Andrew Koenig and Bjarne Stroustrup

[2] The Design and Evolution of C++, Bjarne Stroustrup

[3] More Effective C++, Scott Meyers

[4] Exceptional C++, Herb Sutter

[5] Programming languages C++, ISO/IEC 14882 September 1998