Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS*

ANSI DOC NO: X3J16/92-0120
ISO DOC NO: WG21/N0197
DATE: November 19, 1992
PROJECT: Programming Language C++
REF DOC:
REPLY TO: Jonathan E. Shopiro
shopiro@usl.com

Issues Related to Exception Handling in C++

Abstract

In the process of implementing and testing the Exception Handling feature of C++, a number of questions about its exact definition and interactions with other features arose. This document records some of those questions with discussion and tentative resolutions.

* *Operating under the procedures of The American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001

Issues related to Exception Handling in C++ – Issue 2

Margaret A. Ellis

UNIX System Laboratories
Summit, New Jersey 07901-1400

1. About This Document

This is the second issue of what is expected to be a living document. In the following pages, I present issues that have arisen to date with respect to the exception handling feature in C++ and its implementation in cfront.

Each subsection below is formatted as follows:

— A statement of the issue.

— A resolution section containing a statement of the resolution or an indication that the issue is still open.

— A discussion section containing the electronic mail exchanged on the issue, edited lightly for readability.[1]

The participants in the electronic discussions are

- Martin Carroll of Bell Laboratories
- Margaret Ellis of Unix System Laboratories
- Paul Faust of Hewlett-Packard
- Andy Koenig of Bell Laboratories
- Glen McCluskey of Unix System Laboratories
- Michey Mehta of Hewlett-Packard
- Jonathan Shopiro of Unix System Laboratories
- Bjarne Stroustrup of Bell Laboratories

This remainder of document is organized as follows:

This document will evolve as new issues arise, as resolutions are agreed to, or as resolutions presented here change. Feedback on this document or input for a future issue is welcome; send e-mail to `ellis@usl.com`.

2. Throwing an Exception

2.1 Can a class with ambiguous base classes be thrown?

That is, should this be legal?

---

1. If I have misrepresented or misquoted anyone, I assure you it was inadvertent. Please let me know, so that I can correct the next edition of this document.

```
struct A { ... };
struct B1 : A { ... };
struct B2 : A { ... };
struct C : B1, B2 { ... };

void f()
{
      C c;
      throw c;  // legal?
}
```

2.1.1 RESOLUTION: No, throwing a class with an ambiguous base class is not legal.

2.1.2 DISCUSSION

>From Jon Shopiro:

Here's the way I think it should work.

```
struct A { ... };
struct B1 : A { ... };
struct B2 : A { ... };
struct C : B1, B2 { ... };

void f()
{
      C  c;
      throw c;  // compile-time error, should be okay
}

void g1()
{
      try {
            f();
      }
      catch (C) { ... }
}

void g2()
{
      try {
            f();
      }
      catch (A) { ... }
}
```

The current interpretation gives a compile-time error in `f()`. I think this should be allowed, because if `f()` is called by `g1()` there's no ambiguity. Somebody's programming style may use some class as a non-virtual base of many classes, and this prohibition would be a serious impediment because most objects could not be thrown.

If `f()` is allowed, then we have to say what happens if `f()` is called by `g2()`. There are two possibilities:

1. some kind of error (undefined or `terminate`);
2. treat the ambiguous catch clause as a no-match and keep looking.

I favor (2) because it will eventually find a good match or call `terminate` anyway. It is often the case that when you have base classes of base classes, the user of the derived class may not know all the base classes of an object.

>From Margaret Ellis:

I think it should be legal. C++ allows *potential* ambiguities, which is what this is.

Is the thinking that EH is to be used for building fault-tolerant systems, so EH errors that are not detected until runtime are unacceptable?

>From Bjarne Stroustrup:

We had a long talk about this at the time in the ANSI meeting and agreed that the throw of `C` should be illegal because the catch of `A` would be ambiguous. The alternatives seemed to be

1. that a `C` would quietly not be caught as an `A`
2. that the ambiguity was caught at compile time and turned into an error (`unexpected()`?)
3. that `C` would be caught as some `A` (undefined which `A`, or the first `A` in the construction sequence for `C`) We decided that disallowing throws was safer and easier to understand and that the "restriction" was of no practical importance.

In retrospect I still agree.

We didn't - as I remember it - consider Jonathan's solution (2).

I think I still prefer the resolution we reached - even if I don't take the hassle of re-opening the issue into account.

2.2 Can a class with multiple instances of the same base class be thrown if only one of the base class instances is accessible?

2.2.1 RESOLUTION: No, a class with multiple instances of the same base class cannot be thrown even if only one of the base class instances is accessible.

2.2.2 DISCUSSION

>From Michey Mehta:

```
ARM 15.7:  an object with two uses of a class as a base
ARM 15.7:  cannot be thrown unless both are virtual bases;
ARM 15.7:  otherwise the throw would be ambiguous.
```

Our interpretation of this has always been that you mean "two uses of `A` as a base, both of which are accessible". It should be OK to [throw] this `D` object:

```
    A
Pub/  \Private
  B   C
   \ /
    D
```

>From Margaret Ellis:

Your diagram shows `A` as a virtual base class, in which case throwing `A` would be fine, but I gather you are asking what happens if `A` is not a virtual base?

In that case, I interpret ARM p. 206 to apply:

> Checking for ambiguity is done before access control or type checking. Thus an ambiguity exists when . . . even if only one is accessible from the base class.

This implies that throwing `A` is not allowed.

>From Andy Koenig:

Ordinarily I would agree. However it is already the case that (I think) if you throw a class with a private base class and catch the private base class, it will just slip right on by.

The theory is that you're not supposed to know about your private base classes.

If one extends that theory, it might just make sense to say that if you have only one public instance of a given base class, it's OK to throw.

2.3 What happens when a reference is thrown?

2.3.1 RESOLUTION: A temporary is allocated, the object referenced by the throw argument is copied into the temp, and the search for the appropriate handler is begun.

When the handler is found, if its argument is not a reference type, the local is initialized from the temp. If the handler's local variable is of a reference type, the reference is made to refer to the temp.

The possibly surprising effect of this is that if a reference to a global is thrown, and the handler's local is a reference type, the handler will get a reference to the temporary, not a reference to the global.

Note that a reference can be rethrown after the value stored in the changing temporary addressed by the reference has been changed. The changed value will be propagated in the rethrown object.

2.3.2 DISCUSSION

>From Jon Shopiro:

As I (re-)read the draft standard, throwing an exception is not much like calling a function.

The first thing that happens when an exception (with argument) is thrown is that a temp of the static type of the argument is initialized with the argument. (The case when the argument is of type `T&` is not clear).

I still am not sure what happens when the thrown object is a reference.

```
f(int& ir)
{
        throw ir;
}
```

is the temp an `int` or an `int&`? My current feeling is that the name of an object is just another reference to the object, that is, in this code

```
{
        int  i;
        int& ir = i;
        //  i and ir are completely equivalent here
}
```

which would imply that the temp is an `int`.

>From Andy Koenig:

Jonathan's suggestion: would make it impossible to throw local variables. That is not what we had in mind.

>From Jon Shopiro:

Perhaps I was too terse. In fact treating an object's name as just another reference to the object makes throwing local variables work fine (but throwing a reference to a global might be surprising). For example

```
int  glob;
void  f(int what)
{
      int  loc;
      int& locr = loc;
      int& globr = glob;
      switch (what) {
      case 1: {
            throw loc;
      }
      case 2: {
            throw locr;
      }
      case 3: {
            throw globr;
      }
}
```

In the above (I hope it's right; it's the most complicated program I have written in quite a while) all three throws would be treated alike. That is, an integer temp is allocated, the object referenced by the throw argument (`loc` in cases 1 and 2, `glob` in case 3) is copied into the temp, then the search for the handler is begun.

When the handler is found, if its type is `int`, the local `int` is initialized from the temp. If the type of the handler is `int&`, the reference is made to refer to the temp.

The only surprising result in this interpretation is that if a reference to a global (case 3) is thrown, and the handler type is `int&`, the handler will get a reference to the temp, not the expected reference to a global.

In fact treating the name of a local object as just another reference to the object is also appropriate if we chose the other model of exception throwing (that is, throwing an exception is like calling a function or returning a result from a function), but then instead of the anomaly described above, you get a dangling reference if a local object of type `T` is thrown and is caught by a handler whose type is `T&`. This would be unfortunate, but it can be explained and understood in terms of familiar concepts. (I.e., throwing a local variable (or a reference to a local variable) of type `T` to a handler with type `T&` is like returning a local variable of type `T` from a function whose return type is `T&`.)

>From Andy Koenig:

I'm confused: I can't figure out which of two things Jonathan is saying. The key question (I think) is this: Suppose `v` is a variable of type `T` and you say `throw v` which causes control to pass to a `catch(T& t)` clause. Clearly, `t` is a reference to some object. Is that object `v` itself or a copy of `v`?

I thought that Jonathan was saying earlier that it should be the same object. I said earlier that it should be a copy. Now Jonathan's latest example seems to indicate that he thinks it should be a copy too.

What am I missing?

>From Jon Shopiro:

Perhaps Andy's confusion is due to the fact that there are two possible models (that I am thinking of) for exception throwing, and I have discussed both of them.

One model is that exception throwing is like passing arguments to functions and like returning results from functions. This model leads to dangling references when a local variable of type `T` is thrown

and the catch clause specifies type `T&`.

The other model is that there is an intermediate temporary object which is initialized from the thrown object. This model leads to the anomaly that a thrown reference to a global which is caught by a reference catch clause no longer refers to the global.

The ARM clearly specifies the second model. The reason the first model keeps coming up is that people naturally prefer familiar concepts and function calling is a familiar concept.

The issue of whether the name of a local object is treated differently from a reference to that object is a red herring; they are treated the same.

2.4 Can the name of an overloaded function be thrown?

2.4.1 RESOLUTION: No, the name of an overloaded function (really, its address) can not be thrown.

2.4.2 DISCUSSION

>*From Jon Shopiro:*

The first thing that happens when an exception (with argument) is thrown is that a temp of the static type of the argument is initialized with the argument.

This is pretty clear evidence that the argument must be unambiguous in its own right, so the name of an overloaded function is not an acceptable argument.

2.5 What is the precedence of `throw`?

2.5.1 RESOLUTION: A *throw-expression* is an *assignment-expression.*

2.5.2 DISCUSSION

>*From Glen McCluskey:*

Page 354 has the *throw-expression* being a *unary-expression*; we changed it to *assignment-expression.*

In `gram.y` this looks like:

```
THROW e %prec CM
```

Editor's Note:

```
throw f(a), g(a);
```

should mean

```
( throw f(a) ), g(a);
```

not

```
throw ( f(a), g(a) );
```

2.6 Can a throw appear in a conditional expression?

That is, are these legal?

```
void f()
{
        int x;
        x ? throw : 12;
}
```

```
void g()
{
        int x;
        x ? 12 : throw;
}
```

2.6.1 RESOLUTION: Yes, a throw can appear in a conditional expression.

2.6.2 DISCUSSION

>From Michey Mehta:

We used to reject this, but have changed cfront to *not* reject this. We believe that the ANSI committee will allow both throws and rethrows in conditionals. Please update the test if you agree with our interpretation, or let us know if you disagree.

>From Glen McCluskey:

5.16 prohibits usage of the form:

```
x ? void(0) : 37;
```

because the two arms of the operator don't have the same type.

In terms of types the above case is identical to this and so is illegal.

If ANSI wants to special case throw I won't argue the point, but for now I have to assume this is illegal.

>From Jon Shopiro:

The current draft explicitly allows throws in conditional expressions.

5.16 now has a new paragraph ....

> If either the second or third expression is a *throw-expression*, the result is of the type of the other.

2.7 Are nested throws allowed?

2.7.1 RESOLUTION: Yes. When a nested throw occurs, processing of the previous exception is abandoned and the new exception is processed.

2.7.2 DISCUSSION

>From Glen McCluskey:

Steve mentioned a point at lunch about using signals to capture stuff like `SIGFPE`. For the case below the EH runtime library gets a divide by zero error [which was a bug in the EH runtime at the time of this mail and may not occur with the latest version] in the middle of doing a throw.

I amended the test case to do the equivalent of catching the signal for this and doing a `MathErr` throw. On this nested throw the EH library bails out to `terminate()`.

This might be an issue to track at various levels -- what is the effect of a nested throw, is the signal handling scheme adequate to pick up `SIGFPE`, and so on.

The stack traceback is:

```
program terminated by signal ABRT (abort)
kill() at 0xf775c120
terminate__Fv() at 0xac00
__eh_do_throw__11__eh_markerSFP18__eh_thrown_object() at 0x3dac
__eh_throw__FPvPP13__eh_typeinfoiUi() at 0x3ca0
handle_fpe__Fi() at 0x22b8
_sigtramp() at 0xf7732c4c
destroy_object__FP13__eh_typeinfoPvilT3() at 0x2b48
destroy_object__FP13__eh_typeinfoPvilT3() at 0x2c04
__eh_cleanup_regions__FP19__eh_cleanup_markeri() at 0x2fcc
__eh_do_throw__11__eh_markerSFP18__eh_thrown_object() at 0x3ea4
__eh_throw__FPvPP13__eh_typeinfoiUi() at 0x3ca0
f1__Fv() at 0x23dc
'main() at 0x2598
```

Here is the program:

```
#include <signal.h>

enum StdExc {MathErr};

void handle_fpe(int)
{
      throw MathErr;
}

struct A {
      A() {}
      ~A() {}
};

struct C {
      C(int) {}
};

struct B : public C {
      A a;
      int x;
      int y;
      B() : x(37), a(A()), C(-37), y(47) {}
};

void f1()
{
      B b;
      throw b;
}
```

```
main()
{
        int flag;

        signal(SIGFPE, handle_fpe);
        flag = 0;
        try {
                B b;
                f1();
        }
        catch (B b) {
                B b2;
                if (b.y != 47)
                        return 1;
                flag = 1;
        }
        catch (StdExc) {
        }
        if (!flag)
                return 2;

        return 0;
}
```

What if the EH library uses a function that throws an exception? Suppose my application throws an exception, and while processing it, operator `new()` is called and it in turn throws an exception (out of space).

Is this type of reentrancy supported?

>From Andy Koenig:

I consider it essential to be able to throw and catch exceptions within the (dynamic) scope of another handler. Thus, for example, if I have a function like this (as a limiting case):

```
void f() {
        try {
                throw 3;
        } catch (int) {
        }
}
```

it is important for it to be possible for this function to be called from within an exception handler or from a destructor that itself is called during stack unwinding.

>From Jon Shopiro:

Fully agree with the above.

>From Andy Koenig:

The one we haven't been able to figure out is what to do when stack unwinding throws an exception that itself tries to unwind beyond a handler for the original exception.

>From Jon Shopiro:

This case is where an exception is thrown, the system looks down the stack to find the appropriate handler, finds it, and then starts unwinding the stack, calling destructors as it goes, and one of the destructors (or some function it calls) throws an exception. There are four cases:

1. The handler for the new exception is found before (above) the handler for the exception already being processed.
2. The handler for the new exception is the same as the handler for the exception currently being processed (remember the handler hasn't been entered yet, it has only been found and marked).
3. The handler for the new exception is found after (below) the handler for the exception already being processed.
4. No handler is found for the new exception.

Case 1 is the one discussed and agreed upon above.

I think case 4 is easy to dispose of. Throw an exception for which there is no handler, call `terminate`. `terminate` may not throw an exception (except a case 1 exception) so there is no control path that returns to any part of the execution thread.

For case 3 I think we must deal with the new exception before anything else happens. It doesn't make sense to ignore the new exception and putting it on a shelf to be processed later seems impossibly complex. But then if we handle the new exception that involves unwinding the stack beyond the handler for the exception currently being processed which makes it impossible to resume processing the current exception. The only other choice I can see is to call `terminate` (that used to be my favorite choice).

So my opinion is that in case 3 we should abandon processing the current exception, and start processing the new exception, since calling `terminate` should be a last resort.

To be consistent in case 2 we have to process the new exception first. But then how and when would we get back to processing the old exception? Once we start processing the new exception we are establishing a thread of control that has no obvious interruption point. So I conclude that even in case 2 we should abandon processing the current exception and start to work on the new one.

>From Andy Koenig:

I am very nervous about the possibility of an exception being lost altogether just because a destructor throws some other exception.

>From Jon Shopiro:

Well, true, but if the alternative is `terminate()` losing one exception might not be so bad.

>From Michey Mehta:

The ARM clearly states in 15.6.1c that if a destructor called during stack unwinding tries to exit by using an exception, `terminate` should be called. In other words, the only handlers which are applicable are the ones pushed on *after* the destructor has been entered. If these handlers don't handle the exception, it's bye bye . . . we certainly don't check to see if the handler which handled the original exception can handle the new exception. Thus, it would seem to me that none of the 4 cases listed by Jonathan apply. I must be missing something here . . .

>From Jon Shopiro:

Michey Mehta (HP) pointed out the following quote from the ARM (15.6.1, pg 364).

> . . . when a destructor called during stack unwinding caused by an exception tries to exit using an exception [`terminate()` is called].

Peggy and I were discussing this and the first thing that came out of it was a better understanding of the structure of destructors and what can happen when an exception is thrown within a destructor, even in normal processing.

A destructor consists of a segment of user-written code, followed by compiler-generated calls to the destructors of the sub-objects. Thus the dynamic scope of a destructor of an object includes the destructors of its sub-objects, but the dynamic scope of the user-written segment of a destructor does not include the user-written segment of the destructors of any sub-objects. If an exception is thrown within a destructor, it must be (dynamically) within exactly one of the user-written code blocks. If the exception is to be caught in the dynamic scope of that destructor, it can only be caught in the same code block in which it was thrown. If this occurs, there is no problem and the word "exit" in the ARM is probably used to exclude that case. If the exception is not caught within that code block, it cannot be caught in the destructor of that object or any containing object. I will use the phrase "exits from a destructor using an exception" to refer to that case.

So what should happen when the destructor of a sub-object of an object exits using an exception? It seems clear that the remaining undestroyed sub-objects must be destroyed and if the destructor was invoked on behalf of a delete expression (as opposed to cleaning up a local variable when exiting a scope) the storage must be freed. Otherwise a half-deleted object would be left and it would be impossible to attain a consistent state. This is similar to the problem of destroying only the constructed objects in a stack frame, which can be thought of as a complex object with several sub-objects.

Here is a scenario where exceptions might be thrown by a destructor. Suppose the iostream mechanism throws the exception "clogged_output" when the output fails (e.g., disk full). Now suppose the program looks like this:

```
main() {
      for (;;) {
            try {
                  real_main();  // this function does the work
                  exit();
            }
            catch (clogged_output x)
            {
                  x.plunge();
            }
      }
}
```

Now if the disk fills up and the exception is thrown, you would like to unwind the stack, and enter the handler. Presumably the destructors that are called as the stack is unwound will clean up the state of the program, so that it will be reasonable to try again after plunge() does its repair work.

Suppose also there is a class `X` as follows

```
class X {
      static int  cnt;
      int  id; X() : id(cnt++) { cout << "Hello X#" << id << "0; }
      ~X() { cout << "Goodbye X#" << id << "0; }
};
```

and suppose we have `X` objects all over a program, as local variables and as members of other classes. If the disk fills up and an exception is thrown, the process of unwinding the stack will cause many `X` objects to be destroyed, and presumably each one will cause another exception to be thrown. According to the statement in the ARM, this would cause `terminate()` to be called.

Remember that `terminate()` can only shut down the program; there is no way back once `terminate()` has been called. This is unfortunate because if we could somehow get to the catch clause, the program could continue.

Looking back to the quote from the ARM at the beginning of this note, it raises a serious problem. If a destructor decides it needs to throw an exception, how can it tell if that would lead to a call of `terminate()`? That is, how can a destructor tell if it is being called because a scope is being exited or because the stack is being unwound by the exception mechanism? The answer in general is, it can't.

It seems to me very likely that once a destructor exits using an exception, destructors called as a result of stack unwinding are likely to exit that way also. Therefore there is not much use in allowing a destructor to exit using an exception during normal processing but not allow a destructor to exit using an exception during stack unwinding. Certainly the language would be simpler to define and implement if destructors were not allowed to exit using an exception under any circumstances.

If destructors are allowed to exit using an exception during stack unwinding, what then? The answer is that the exception currently being handled must be abandoned: for an explanation, stay tuned.

2.8 What happens if a rethrow occurs outside the dynamic context of a handler?

2.8.1 RESOLUTION: The behavior of a rethrow outside the dynamic context of a handler is undefined. Our implementation will call terminate.

2.8.2 DISCUSSION

>From Glen McCluskey:

The ARM doesn't quite say what happens when you do rethrows at random, but the existing implementation calls `terminate()`, which seems like a good idea to me.

```
extern "C" {
      void exit(int);
      void puts(char*);
};

typedef void (*PFV)();

PFV set_terminate(PFV);

void t()
{
      puts("terminate");
      exit(1);
}

main()
{
      set_terminate(t);

      throw;
}
```

>From Jon Shopiro:

If you `throw;` outside the dynamic context of a handler, the result is undefined.

>From Glen McCluskey:

Suppose you are off in your own little world and you say:

```
throw -97;
```

You don't know about any handlers that might exist, but you do know that `terminate()` can be used to avoid an `abort()`.

Wouldn't the same apply to:

```
throw;
```

You may be in a function called from within a handler, or then again you may not be. The key point is that you shouldn't have to know.

>From Jon Shopiro:

I beg to differ. If you say

```
throw -97;
```

there had better be a handler. If not it's an error. The `terminate()` function exists to let you shut down gracefully, that's all.

Typically errors of this sort are undefined and `terminate()` was put in the language to cater to some nervous nellies.

If you say

```
throw;
```

that means re-throw the current exception, and if there's no current exception, it's an error. There's no definition for what happens then in the documents, therefore it's undefined.

C++ has undefined so that implementations do not have to check for every possible error, which would be prohibitive.

The idea that you shouldn't have to know is wrong. You do have to know. Exceptional behavior is part of the interface of a function or library and if you violate the interfaces, bad things will happen.

>From Glen McCluskey:

I don't feel strongly about this.

I would, however, suggest we inform HP of this decision and have them call `abort()` instead of `terminate()` in this case.

>From Margaret Ellis:

15.6.1 says that `terminate` will be called "when the exception handler cannot find a handler for a thrown exception." Now, you may say that a rethrow when no handler is active has no exception to throw and therefore isn't covered by this statement, but I think that the spirit of the design pretty clearly indicates that a rethrow with no exception to rethrow should wind us up in `terminate`.

I would also say that 15.6.1 should have some words added to explicitly cover the case of a rethrow with nowhere to go.

>From Jon Shopiro:

I would prefer to avoid saying what happens when this or that error occurs. It puts a burden on the implementor with little or no benefit to the user.

2.9 How should a throw of a pointer to a function be treated?

2.9.1 RESOLUTION: A throw of a pointer to a function should be allowed (with a warning under `+w`).

2.9.2 DISCUSSION

>From Michey Mehta:

The ARM says:

A pointer to function may be converted to `void*` provided a `void*` has enough bits to hold it.

This is fine, because any attempt to perform such a conversion on a system where `void*` has insufficient bits can be rejected at compile time. However, with EH this conversion takes place at runtime.

```
try {
      throw some_function_pointer;
}
catch(void*){
      //It may or may not be caught here
}
```

This means that this program will have different behaviors at runtime depending on where you run it. This looks undesirable. Runtime behavior of throw should be deterministic ... i.e. you should be able to answer the question "Does a `catch(X)` catch a throw of `Y`?" in a platform independent way. We have previously discussed a somewhat similar problem to do with `size_t`.

What should our implementation do about throwing function pointers?

1. Quietly skip a `catch(void*)` if the thrown type is function pointer that has more bits than a `void*`. (We do this today)
2. Reject a throw of a function pointer on systems where `void*` is smaller than a function pointer. This seems drastic.
3. If `void*` is smaller than a function pointer then allow the throw, but produce a warning at compile time.

>*From Margaret Ellis:*

A warning seems appropriate. Michey leaves open the question of what happens at runtime. I think that `catch(void*)` should be passed because the conversion is not legal. `A catch(...)` or a `catch(some_pointer_to_function_type)` may be found. If not, `unexpected` or `terminate` will be called.

>*From Andy Koenig:*

[...] since the ability to convert a function pointer to `void*` is at the option of the implementation, I still vote for quietly ignoring the existence of such a conversion across the throw/catch interface.

Warning about every throw of a function pointer is equivalent for many users to prohibiting throws of function pointers. That is far too drastic, as one can reasonably throw a function pointer and catch one of the same type. It is much harder to imagine throwing a function pointer and catching `void*` because I *know* that will fail on many, many systems. In particular, function and data pointers are completely different beasts under MS-DOS.

>*From Margaret Ellis:*

Considering that many applications must compile warning-free, I suppose Andy's right.

Nevertheless, a warning under `+w` seems appropriate.

2.10 Should thrown classes have external linkage?

2.10.1 RESOLUTION: Yes, thrown classes should have external linkage.

2.10.2 DISCUSSION

>*From Michey Mehta:*

Section 3.3 of the ARM talks about classes with internal linkage ... they should have no out of line member functions, no static data members, and not be used in any externally visible variable or object declarations.

If such a class object is thrown, should it be caught in another file which contains a class of the same name (but which has internal linkage)? I think it should. I would suggest modifying section 3.3 to indicate that a class which is thrown or caught (and all base classes) will get external linkage.

Arguably, the ARM already says this, because it talks about declaring a temporary. I suppose you could view this temporary as being an externally visible object, thus disqualifying the class from being a class with internal linkage. However, it's better to be explicit about this in the language spec.

>From Margaret Ellis:

I think Michey is right - 3.3 should be modified to explicitly specify that a class will not have internal linkage if any of the following is true:

— an object of the class type is thrown,

— the class type appears in a catch expression, or

— the class type appears in a throw specification list.

>From Bjarne Stroustrup:

The internal linkage rule is weaselwording to avoid clashing with pre-existing C code. It might be better to review 3.3, but I think - unless we introduce a stronger notion of internal class - that the EH mechanism should assume that any class thrown has external linkage. Has the possibility of throwing a local class object been considered? Example:

```
static void f()
{
        struct X { };
        throw X();
};
```

I don't think it will cause any real problems (we considered such perversities in the context of RTTI). It simply will not be caught (the plain name of a class isn't sufficient identification of a type - scope must be considered).

>From Margaret Ellis:

Yes. We decided that a warning under `+w` was appropriate for a throw of a local class.

Note that a throw of a local class will be caught by `catch(...)`, if present.

2.11 Must a copy be made of the thrown expression on a rethrow?

2.11.1 RESOLUTION: No, a copy need not be made of the thrown object on a rethrow.

2.11.2 DISCUSSION

>From Michey Mehta:

The ARM says that a copy must be made on a throw. What about a rethrow? We do not currently make one on a rethrow because we already made a copy when the original throw took place. Is this right?

>From Margaret Ellis:

Yes. The text that calls for initializing the temporary refers to the operand of the throw:

> A *throw-expression* initializes a temporary object of the static type of the operand of throw and uses that temporary to initialize the appropriately-typed variable named in the handler.

For a rethrow, there is no such operand.

Further, the annotation at the end of 15.2 assumes that the EH RTS would logically use the same temporary object for the rethrow:

> This definition of `throw;` implies that the exception handling mechanism needs a concept of a "*current exception.*" It also implies that although an exception is considered handled on entry to a handler, the current exception cannot be deallocated until exit from its handler.

3. Handling an Exception

3.1 Should the implementation warn or generate a hard error for the appearance of a masked catch clause?

3.1.1 RESOLUTION: The appearance of a masked catch clause is an error.

3.1.2 DISCUSSION

>From Paul Faust:

In your mail message, there are two references to a condition where two identical types found in a catch sequence should be treated as warnings.

Editor's Note: Paul's quotation from above issue deleted here to save space.

We have been treating this as a hard error. The rationale for this stems from section 15.4:

1. The handlers for a try-block are tried in order of appearance. It is an error to place [a handler] for a base class ahead of a handler for its derived class since that would ensure that the handler for the derived class would never be invoked.
2. If present, a `...` handler must be the last handler for its try-block.

In both the above cases, the rationale for treating these as errors is that they would "mask" other catch blocks. Similarly, the `size_t` catch block would mask the `unsigned int` catch block and should also be treated as an error. Additionally, for the program written as:

```
try {}
catch (int) {}
catch (int&) {}
catch (const int) {}
```

and its various permutations, the other two catch blocks would be "masked" by the first specified catch block.

>From Margaret Ellis:

Paul's analysis makes sense. Do you agree that the intention of the 15.4 is that the appearance of a "masked" handler is an error (if so, 15.4 should be made more explicit)?

>From Glen McCluskey:

Yes.

>From Jon Shopiro:

I agree (?) that redundancy in exception-specifications should be accepted without comment, but masked catch clauses (for a single try block) should result in hard errors.

Here's an example:

```
try {
      // no way to throw an exception here
      try {
            // potentially throw an exception here
      }
      catch (int i) { ... }
            // no way to throw an exception here
}
catch (int i) {  ... } // legal, even though effectively masked
```

It occurs to me that when control flows off the end of a try-block (that is, no exception is thrown), it's awfully difficult to see where the next statement to be executed is.

>From Andy Koenig:

I am very nervous about making such things errors because doing so might seriously restrict what can be done with templates. For example, suppose there's a global exception called `Error` and we're in a template:

```
template<class T, class E> class Foo {
      // ...
      void f() {
            // ...
            try { g(); }
            catch (E) { /* ... */ }
            catch (Error) { /* ... */ }
            // ...
      }
      // ...
};
```

The idea is that the second template argument to this class says which exception to catch when calling `g`. Now, if exception handlers are allowed to duplicate each other, the user can "turn it off" by passing `Error`, thus masking the other catch of `Error`.

Again, I'm not sure whether this is an argument for or against it, but it deserves thought.

>From Margaret Ellis:

Paul, you are right. Bjarne was imprecise when he said "warning," and I didn't catch it. The appearance of a masked handler is an error. The ARM and the ANSI draft should be made more explicit on this point.

3.2 Does the presence of a linkage specification affect the handlers that can catch (the address of) a function?

Should this throw:

```
extern "C" {
      void f(int);
};

void g()
{
      throw f;
}
```

be catchable by:

```
catch (void (*)())
```

3.2.1 RESOLUTION: No, the type of a function is not affected by a linkage specification.

3.2.2 DISCUSSION

>From Jon Shopiro:

No. Even though it's extern C, the function still has a C++ type.

>From Margaret Ellis:

Right. The annotation at the top of ARM p. 119 addresses this. Probably the reference manual proper and the standard should state this explicitly.

3.3 Can an incomplete type appear in a catch clause?

3.3.1 RESOLUTION: No, an incomplete type can not appear in a catch clause.

A pointer or a reference to an incomplete type may appear in a catch clause, however.

3.3.2 DISCUSSION

>From Margaret Ellis:

We have discussed the question of whether incomplete types can appear in throw specification lists and have decided that they can. Someone (I forget who - sorry!) has raised the question of whether incomplete types can appear in catch clauses.

Certainly the following case has to be illegal:

```
struct A;
void f() {
      try {
            // some stuff
      }
      catch (A a)  // how do we initialize A?
      {
            // ...
      }
}
```

but what about this?

```
struct A;
void f() {
      try {
            // some stuff
      }
      catch (A)  // no variable to initialize
      {
            // ...
      }
}
```

>From Jon Shopiro:

If it isn't hard to implement (it shouldn't be), make it legal.

>From Michey Mehta:

The following is rejected today (for good reason):

```
struct A;
void f(A){}
```

Since catch clauses declarations are somewhat analogous to parameter declarations, why bother allowing a `catch(A)`?

I am in favor of prohibiting the use of incomplete types (and pointers to incomplete types) in catch clauses, specification lists, and throw statements. An alternative is to have clear verbiage in the RM about exactly what it means to use incomplete types in these contexts.

>From Andy Koenig:

I'm dubious about throwing incomplete types and even more dubious about catching them. For one thing, we've been talking about various restrictions on things like virtual base classes; those restrictions can't be checked on incomplete types.

>From Paul Faust:

I agree. Allowing incomplete types bypasses two static diagnostics that are made today.

First, ambiguous base classes cannot be detected. For example, in `file1`:

```
D* d;

try {
  throw d;
}

catch (A* a1) {}
```

in `file2`:

```
struct A {};
struct B : public A {}
struct C : public A {}
struct D : public A, public B {}
```

when doing the catch in `file1`, which `A` should be chosen, `B::A` or `C::A`??

The second check that wouldn't be performed is the masking of catch blocks:

For example, in `file3`:

```
try { foo() }
catch (A) {}
catch (B) {}
catch (C) {}
catch (D) {}
```

catch blocks for `B`, `C`, and `D` will never be executed.

As has been pointed out earlier, similar problems happen with assignments between pointers and the passing of pointer arguments; no address arithmetic will be performed. The tradeoff of disallowing incomplete classes for assignment and argument passing in favor of "correctness " would be a painful translation of C applications to C++.

Also, it has been pointed out that incomplete types prevent header file proliferation.

For eh, migration is not an issue; it will be all new code. Hopefully, the objects being thrown and

caught will be new types defined specifically for eh and do not have such a scope that they have to bring in the world to define them.

With a new feature such as this, we should limit the number of surprises for users.

>*From Bjarne Stroustrup:*

Andy says

> I'm dubious about throwing incomplete types and even more dubious about catching them. For one thing, we've been talking about various restrictions on things like virtual base classes; those restrictions can't be checked on incomplete types.

We clearly can't throw incomplete types. However, exactly because all checking for potential ambiguities is done at the throw point I see no problem with checking at a catch point for an incomplete type.

However, I do see reason to prohibit them.

I would hate to have to explain why

```
struct S;

//...

catch (S) { ... }
```

was legal and

```
catch(S a) { ... }
```

wasn't. We cannot declare arguments of incomplete types:

```
f(S) { ... }
```

is no more legal than

```
f(S a) { ... }
```

and I think we should maintain that parallel.

>*From Bjarne Stroustrup:*

I think that incomplete types should be allowed in specification lists because one purpose of a specification list is to specify what may pass through untouched. Since the exception is untouched it need not be known.

This leads to an ugliness in the specification.

```
void f() throw(A) { ... }
```

means

```
void f()
{
      try { ... }
      catch(A) { throw; }
      catch(...) { unexpected();
}
```

but unless A is defined we can't actually write that.

We could remove the ugliness by allowing

```
catch(A)
```

where A is incomplete, but that is the only argument I can see for allowing that and - as mentioned in my previous mail - I think allowing that would lead to some minor confusion.

3.4 When is an exception considered "handled?"

3.4.1 RESOLUTION: An exception is considered "handled" when one of the following occurs:

1. A handler for the exception is invoked.
2. `terminate` is invoked.
3. `unexpected` is invoked.

3.4.2 DISCUSSION

>From Michey Mehta:

We have been having some discussion about exactly what exception to throw when we see a rethrow.

The ARM says that a rethrow will throw the exception being "handled", and may only appear in a handler, or in a function directly or indirectly called from a handler. Since the example on page 365 contains a rethrow in `pass_through` which is called by `unexpected`, I conclude that both `unexpected` and `terminate` are considered to be "handlers" which handle the exception that caused them to be entered.

To help us implement this correctly, here is my wording for exactly what a rethrow should do:

> You always rethrow the exception currently being "handled". An exception is being "handled" if any of the following are true:
>
> 1. A handler for the exception is active on the stack.
> 2. There is an invocation of `terminate` on the stack.
> 3. There is an invocation of `unexpected` on the stack.
>
> Note that after an exception is thrown, it is considered to be "handled" only after it reaches one of three states described above. In particular, it is *not* a candidate for a rethrow by:
>
> — Any rethrows in the copy constructor at the throw point
>
> — Any rethrows in destructors invoked by the runtime for object cleanups
>
> — Any rethrows in delete operators invoked by the runtime for freeing memory
>
> — Any rethrows in the copy constructor at the catch point

>From Margaret Ellis:

Note that `terminate` cannot rethrow, nor can a destructor called during stack unwinding.

>From Jon Shopiro:

I think the easiest point of view is that `unexpected` _is_ a handler, except that it cannot return. So is `terminate`, except that it cannot return or throw.

If a destructor tries to rethrow, terminate.

3.5 Should a partially constructed heap object be cleaned up and its memory freed on an exception?

3.5.1 RESOLUTION: Yes, a partially constructed heap object should be cleaned up and its memory freed on an exception.

3.5.2 DISCUSSION

>From Michey Mehta:

[. . .] we have assumed all along that a partially constructed heap object must be cleaned up and memory deallocated upon an exception. However, I don't think this assumption is documented in the ARM or the issues document, although it is alluded to in the issues document.

3.6 What happens if an exception occurs during delayed initialization of statics?

3.6.1 RESOLUTION: OPEN

3.6.2 DISCUSSION

>From Michey Mehta:

If an exception occurs during initialization of static variables, we just call `terminate` (which may choose to call `exit`). However, the ARM permits static initializations to be delayed until the first use of an object in the compilation unit ... this is how some shared library implementations work.

Given that initializations may be delayed, it is possible that the execution of the main program has already started, and at some later point an exception due to static initialization occurs. What do we do with the stack?

— There may be handlers on the stack. I would think these handlers should not be examined, since they were never meant to be handlers for exceptions in static initialization.

— There may be destructible objects on the stack requiring cleanup. Should these be cleaned up?
Answer 1: Yes.
Rebuttal: This is inconsistent with other kinds of unhandled exceptions where stack objects are not cleaned up.
Answer 2: No.
Rebuttal: If a programmer carefully writes his application so that that there is a `catch(...)` in `main`, and `terminate` calls `exit`, then he has a fighting chance of getting all objects cleaned up (thus freeing any critical resources) in the event of an unhandled exception. But, if stack objects are not cleaned up when an exception occurs during a delayed initialization, then there is no programmatic way of ensuring that all objects are cleaned up.

Note that in an implementation which supports delayed static initialization there are two classes of frames on the stack (from the point of view of EH when an exception occurs in static initialization). The bottom of the stack are those frames which represent the execution of the main program. The top of the stack are those frames which are participating in static initialization; any handlers and cleanup objects in these top frames should always be obeyed. My questions above pertain to handler/cleanup objects in the bottom frames.

>From Margaret Ellis:

If an exception thrown during static initialization were treated the same as any other exception (that is, if there is a handler on the stack, invoke it), then we could run into a violation of the guarantee in 3.3:

> The initialization of nonlocal statics in a translation unit is done before the first use of any function or object defined in that translation unit.

(Suppose the handler calls something from the translation unit for which static initialization failed.)

If there are indeed two classes of frames, then should objects on the stack be cleaned up? I say yes.

>From Jon Shopiro:

I think that static initialization is a separate thread and can be thought of as having its own stack. I think a reasonable approach in case static initialization throws an unhandled exception is to note that the static initialization thread has failed and to throw a special exception (EH_STATIC_INIT_FAILURE) in the main thread.

Suppose I have a function f() which is in a shared library, and calling that function causes the library to become attached and its statics inited.

```
try {
      f();
}
catch (EH_STATIC_INIT_FAILURE) {
      printf("couldn't init f's library0);
      throw EXIT;
}
```

>From Margaret Ellis:

Do you propose that this become part of the language def'n, or an implementation detail? The former would seem more useful for library implementors.

>From Jon Shopiro:

Part of the language definition, to be sure. The committee hasn't really come to grips with the idea that statics in libraries may be initialized after the beginning of `main()`, so it will be a while before they get to this.

4. Throw Specifications

4.1 Must all throw specifications on the definition and declarations for a given function agree?

4.1.1 RESOLUTION: Yes, all throw specifications on the definition and declarations for a given function must agree.

4.1.2 DISCUSSION

>From Michey Mehta:

I have heard the suggestion that the spec on a declaration may mention more exceptions than the spec on a definition (so that the declaration does not always have to change when the definition does). We currently check to make sure that they are exactly the same (although the order is which the exceptions are mentioned may be different). Is this OK?

>From Jon Shopiro:

Suppose a function is to have an exception specification. There may be several declarations of the function in a program and one definition. Where does the exception specification go? If it can appear in more than one place, which version rules? The exception specification is not part of the function's type, but it figures prominently in its semantics. This argues for putting the exception specification on the function definition. However, the exception specification seems to be important to the function's interface with the rest of the program. This argues for putting it on the function declaration.

Here are some possible resolutions, with pros and cons, as I see them:

1. The (functionally) identical exception specification must appear on all declarations and the definition of the function. (By functionally identical I mean reorderings and repetition of classes are not considered a difference, so `throw (X,Y)` is the same as `throw (Y,X,Y)`.

   This is the simplest to state and understand. It is hard to check between compilation units and a pain in the neck for the programmer.

2. Omitted. Too complicated.
3. Ditto.

>From Andy Koenig:

I'm pretty sure [Jonathan's #1 is] what our EH paper says.

Personally, I wouldn't mind relaxing it slightly to say that every declaration must mention every exception that appears in the definition but is allowed to mention others as well.

>From Margaret Ellis:

I don't like Andy's idea of allowing other exceptions to appear on function declarations. Presumably, the function will be allowed to throw only the types listed in the throw spec on the function definition. And if it throws anything else, `unexpected` will be called. Thus allowing declarations to list other types in the throw spec would be misleading.

I would like Jonathan's proposal relaxed to say that the throw spec on all declarations must match that on the definition OR be absent. Thus the lazy programmer could type just

```
void f();
```

instead of

```
void f() throw (long list of types);
```

One would hope that a declaration appears in a header file somewhere, and that it has the throw spec.

>From Andy Koenig:

Peggy says "Thus allowing declarations to list other types in the throw spec would be misleading."

Not really. If I have a declaration like

```
extern void foo() throw (X,Y);
```

that means that if I call `foo`, I should be prepared to deal with exceptions `X` and `Y`. If it actually turns out that `foo` is incapable of throwing `Y`, there's no harm done.

Here is a potentially important reason to allow the kind of permissive mismatch I suggested. Suppose I am supplying a library with a function that might throw `X`. Suppose further that I give you a header file that declares the function as throwing `X`.

Now, let's say that I decide I want to change my function eventually so that it might also throw `Y`. If the header file must exactly match what's actually in the library, then there's no way you can produce a binary that can be linked with both versions of my function.

On the other hand, with the relaxation I suggested, I can give you a new header file immediately and then wait a decent interval to let the old header files die out before changing my library.

>From Margaret Ellis:

Andy says "If I have a declaration like

```
extern void foo() throw (X,Y);
```

that means that if I call `foo`, I should be prepared to deal with exceptions `X` and `Y`. If it actually turns out that `foo` is incapable of throwing `Y`, there's no harm done."

Suppose that `Y` is what my system `malloc` throws when it is out of space. Looking at this declaration, I think that if `malloc` fails somewhere down in `foo` or a function called by `foo`, I can catch `Y` and handle it. In fact, I can't catch `Y` - because my handler won't be invoked;

`unexpected` will be called. That's why Andy's proposed relaxation is misleading.

Andy's suggested use for relaxed throw specs to facilitate migration is nice, but I don't think it outweighs my objection.

>From Andy Koenig:

Peggy says: "Suppose that `Y` is what my system `malloc` throws when it is out of space. Looking at this declaration, I think that if `malloc` fails somewhere down in `foo` or a function called by `foo`, I can catch `Y` and handle it. In fact, I can't catch `Y` - because my handler won't be invoked; `unexpected` will be called. That's why Andy's proposed relaxation is misleading."

Consider:

```
void g() throw (X)
{
        // ...
}

void f() throw (X,Y)
{
        g();
}
```

If `f` calls `g`, `g` calls your function, and your function throws `Y`, you are out of luck anyway. Thus I claim that if you are called by a function that claims it might throw `X` and `Y`, that is no guarantee that you can throw anything at all successfully.

>From Jon Shopiro:

I think "misleading" and "no guarantee" are good descriptors for this whole exception specification business.

I think allowing the exception specification to vary among the declarations and definition of a function makes it even worse.

>From Bjarne Stroustrup:

[Jonathan's proposal] (1) is my understanding.

In the USENIX EH paper the section "Type Checking of Exceptions" starts:

> If a list of exceptions is specified for a function, it behaves as part of the function's type in the sense that the lists must match in the declaration and definition. Like the return type, the exception specification does not take part in function matching for overloaded functions.

Naturally, we could define "matching" as a subset property, but I suspect that there wouldn't be too much advantage for that because the list of exceptions is not part of the type in the safe-linkage sense. In fact, the reason that the list of exceptions is not part of the type in the safe-linkage sense is that if it were then the linker would have to understand about subsets of exceptions. My intent was that "matching" meant "with the same set of exceptions" as Jonathan states.

4.2 Can a class with ambiguous base classes be on a specification list?

That is, is the throw specification on `bar` legal?

```
struct A { ... };
struct B1 : A { ... };
struct B2 : A { ... };
struct C : B1, B2 { ... };
```

```
void foo (C* cp)
{
      throw *cp;   //error according to ANSI
}

void bar () throw(C);   // legal?
```

4.2.1 RESOLUTION: No, a class with an ambiguous base class can not appear in a throw specification.

4.2.2 DISCUSSION

>From Jon Shopiro:

I would generate a warning rather than an error, although an error would be okay too.

>From Bjarne Stroustrup:

I think it would be best to disallow that.

4.3 Can a derived class of a class on a throw specification list also appear in that same throw specification list?

4.3.1 RESOLUTION: Yes, a derived class of a class on a throw specification list can also appear in that same throw specification list.

4.3.2 DISCUSSION

>From Michey Mehta:

If a class on a spec list is a derived class of another class on the spec list is this OK? Warning? Error?

>From Jon Shopiro:

Okay. Even if the same class appears twice in an exception spec list.

4.4 Can function that lists a pointer to a base class in its throw specification list also throw a pointer to a derived class of that class?

4.4.1 RESOLUTION: Yes, a function that lists a pointer to a base class in its throw specification list can throw a pointer to a derived class of that class.

4.4.2 DISCUSSION

>From Michey Mehta:

Page 363 of the ARM says that a function that may throw an exception of a class `X` may throw an exception of any class publicly derived from `X`. What about pointers? If a function claims it throws a "base*", can it also throw a "derived*"?

>From Jon Shopiro:

Yes.

4.5 Can a reference appear in a throw specification list?

4.5.1 RESOLUTION: Yes, a reference may appear in a throw specification list.

4.5.2 DISCUSSION

>From Michey Mehta:

Does a reference make sense in an spec list (i.e `f()throw(X&);`). I would think not.

>From Jon Shopiro:

There is no difference between throwing an object or a reference, so I would make no distinction between having an object type and a reference to that type in a specification list.

4.6 Can a type appear more than once in a throw specification list?

That is, is this declaration legal:

```
void baz() throw(A,A,A);       // legal?
```

4.6.1 RESOLUTION: Yes, duplicate types are to be allowed in throw specification type lists.

4.6.2 DISCUSSION

>From Margaret Ellis:

Note that if I am trying to write portable code I might write

```
void boing() throw(type1, type2);
```

and that `type1` and `type2` might be machine-dependent types that are the same on some machines but different on others.

Also, allowing the same type to appear more than once in a throw specification might facilitate machine-generated code.

>From Bjarne Stroustrup:

I don't think replication in throw-specs is disallowed, and the reasons you give for not disallowing it seem fine to me.

>From Glen McCluskey:

If you allow type duplication, then it's a question of supporting sophisticated users while letting stupid mistakes through.

>From Paul Faust:

Assume the following programs:

```
unsigned int a;
size_t b;
extern int c;

void A() throw (size_t,unsigned int) {
      if (c)
            throw a;
      else
            throw b;
}

void B() {
      try {
            A();
      }
      catch (unsigned int) {
            printf("caught in B0);
      }
}
```

```
void C() {
      try {
            B();
      }
      catch (size_t) {
            printf("caught in C0);
      }
}
```

Further assume that on system 1 `size_t != unsigned int`.

1. `C == 0`
   prints caught in B
2. `C != 0`
   prints caught in C

On system 2, `size_t == unsigned int`.

1. `C == 0`
   prints caught in B
2. `C != 0`
   prints caught in B !!!!

This undesired behavior could have been avoided with a diagnostic from the compiler on system 2!!!

>From Martin Carroll:

I believe he's reversed some cases here, but no matter. Why does he consider this undesired behavior? Seems perfectly ok to me. If I say

```
catch(size_t)
```

I mean "catch whatever `size_t` is `typedef`'d to." If I say

```
catch(unsigned int)
```

I mean "catch `unsigned int`, and whatever else may be `typedef`'d to it."

>From Margaret Ellis:

The "undesirable" aspect I see is confusion on the part of users.

If users throw `size_t`, have a handler for `size_t`, but the handler for `unsigned long` (or whatever) is invoked, I GUARANTEE you we'll see MRs.

>From Bjarne Stroustrup:

Maybe C++ will get a real `size_t` type, but that will not help the problem in general. People will define their own `typedefs`.

People always wants `typedefs` to be types, except for the people who don't want them to be types, of course. We have no choice but to accept `typedefs` according to the language definition.

I think this implies that accepting synonyms (and other repreated occurrences of type names) in throw specifications is a good idea:

```
f() throw(size_t, unsigned int);
```

On the other hand it is easy to warn against multiple catches for a single type:

```
try {
      // ...
}
catch(size_t) {
}
catch (unsigned int) {
}
```

>From Margaret Ellis:

Bjarne's warning will catch some possible errors, but no warning would be produced for Paul Faust's example. In Paul's example, the handlers were in separate functions.

Editor's Note: Bjarne and I both should have said "error" here, not "warning."

>From Andy Koenig:

There is already an informal proposal before the ISO committee to make it possible (somehow) to create 'new' integral types. This is much more important in C than C++ because of overloading.

If that proposal, or something like it, were adopted, it would solve the problem of throwing and catching `size_t` as well. If it isn't, I would take that as a sign that people don't consider the problem worth worrying about.

4.7 Can an incomplete type appear in a throw specification list?

For example, should this be legal:

```
struct A;
void f() throw(A) { }
```

4.7.1 RESOLUTION: Yes, an incomplete type can appear in a throw specification list.

4.7.2 DISCUSSION

>From Glen McCluskey:

This is accepted without complaint. I think it should be illegal so as to give implementations maximum flexibility. This is separate from the pointer and reference cases.

>From Jon Shopiro:

Agreed.

>From Margaret Ellis:

What if I change Glen's example ever-so-slightly:

```
// file1:
struct A;

void g() throw(A);
void f() throw(A)
{
      g();
}
```

```
// file2:
struct A{};

void g() throw(A);
{
      A a;
      throw a;
}
```

Here `f` merely propagates a throw of an `A`. Does the compiler have to know the size of an `A` to generate code for `f`? (I think not - the throw spec is merely informative; the tossed `A` will be handled in the EH runtime routines.) So `f` knows there's a type `A`, but it doesn't need to know `A`'s size.

I think this should be legal.

Editor's Note: Sometime later, the issue came up again independently. Glen and Jonathan both changed their positions.

>From Michey Mehta:

Yet another specification issue:

Can you say `X*` in a throw list, where `X` is an incomplete class?

>From an implementation point of view this is hard to implement, since without seeing the class we must assume that it has an externally visible typeinfo which we can reference. If the typeinfo subsequently turns out to be static (because it has no `vtbl` or has a static `vtbl`) then you'll get an unresolved symbol.

But, putting the implementation hassle aside, do you see a good language argument for allowing or disallowing this?

>From Glen McCluskey:

I think this should be legal, because it fits with the model employed in the rest of the language.

It seems to me that they don't need to know much about the typeinfo stuff except that which is required to identify it. In other words, if you threw `A*` then `B*` cannot catch it.

>From Margaret Ellis:

This should be legal for consistency with the rest of the language.

>From Paul Faust:

I can understand the consistency issue in other circumstances. . . It helps users who are migrating from C to make no code changes. However, specifications are new to the language and will all be new code. Per previous mail, we will be detecting classes in throw lists that are ambiguous. If the class is incomplete, this case can not be caught. I believe we would be doing users a disservice.

>From Margaret Ellis:

The language already allows certain things to slip by with incomplete types that would be caught if complete type info were available. This cast, for example, is legal:

```
class A;
class B;
A* ap;
B* bp = (B*)ap;
```

The value of pointer will not be changed. If it turns out that `B` is a base class of `A` other than the first base class, then the result of the cast will be incorrect. If `A` is a base class of `B`, the cast is bogus. In neither case will the error be caught at compile time.

>From Jon Shopiro:

The reason people use incomplete classes is to avoid header file explosion. I think this is a good reason and they should be allowed to keep doing it.

4.8 Where can a throw specification appear?

4.8.1 RESOLUTION: A throw specification can appear only in a function declaration or a function definition and only for the function being declared or defined. In particular, it may not appear within an argument list or in a typedef.

4.8.2 DISCUSSION

>From Glen McCluskey:

It's not clear from page 363 whether this should be legal or not:

```
void f(void (*)(int) throw(double))
{
}
```

>From Jon Shopiro:

Exception specifications are checked at run time, not compile time, and they are not part of the function's type.

Since the throw specification is not part of the type, if it is allowed in this context (and typedefs) it would be ignored. I think it is better to disallow it so as not to give the programmer false expectations, but I haven't checked the grammar.

>From Margaret Ellis:

The grammar in the ARM doesn't take this into consideration.

At 1st printing, EH (and PT) were experimental, and thus were not integrated into the grammar. When the "experimental" status was lifted, the fact that they needed to be incorporated into the grammar escaped the notice of anyone who might have done something about it (Bjarne and me) or anyone who might have nudged us to do it.

5. The `terminate` and `unexpected` Functions

5.1 What happens when a thrown exception is not handled?

5.1.1 RESOLUTION: No cleanups should take place; `terminate` should be called.

If an unhandled exception occurs while constructing static objects, call `terminate`. If `terminate` then calls `exit`, any fully constructed or partially constructed statics should be destroyed.

If an unhandled exception occurs while destroying static objects, call `terminate`. If `terminate` then calls `exit`, try to destroy any remaining static objects. Do not try again to destroy the object that caused the exception.

5.1.2 DISCUSSION

>From Michey Mehta:

Should any cleanups occur when a thrown exception is not handled?

Our interpretation is that no cleanups should take place; the `terminate` routine should be called.

>*From Jon Shopiro:*
Agreed.

Programs that really care about there not being any unhandled exceptions will wrap everything (if not `main` itself) in a try block with a default handler, so all exceptions will be handled. Programs that don't do this will take their lumps.

>*From Michey Mehta:*
What if `terminate` calls `exit`?

A user defined `terminate` routine may choose to call `exit`. The current `exit` routine attempts to destroy non local static objects. Should we continue to do this? Should we provide another library routine which will clean up any local objects?

>*From Jon Shopiro:*
`exit()` should clean up statics, as it does when called from anywhere else. If the user wants to leave statics alone and dump core, the program can call `abort`.

If the user wants locals cleaned up, then an exception that is caught (perhaps by a default handler in `main`) should be thrown.

It might be a good idea to provide a new library function that would leave local objects alone and stop the program without dumping core. (I wouldn't be at all surprised if ANSI defines such a function, along with a few others.)

>*From Michey Mehta:*
I assume that if an unhandled exception propagates past the point of the original `terminate` call, we would call `abort` immediately.

>*From Jon Shopiro:*
Yes.

>*From Michey Mehta:*
What about unhandled exceptions during the construction of static objects?

A user can always force everything to be cleaned up properly by putting a `catch(...)` in the main program. But this catch clause does not handle an unhandled exception which occurs during the construction of static objects which occurs before `main` is entered. How would a user specify that all global objects constructed so far should be cleaned up?

>*From Jon Shopiro:*
The user can put a try block with a default handler that calls `exit()` in each constructor that may be invoked for a static object.

>*From Michey Mehta:*
We could say that if the `terminate` routine calls `exit` any global objects (and partially constructed global objects) will be cleaned up.

>*From Jon Shopiro:*
Yes.

>*From Michey Mehta:*
A similar problem is what to do with an unhandled exception during the destruction of static objects.

>*From Jon Shopiro:*

If `terminate` calls `exit`, try to destroy any remaining static objects. Do not try again to destroy the object that caused the exception.

>From Michey Mehta:

Static objects are currently constructed before the users main program is entered, and destroyed when `exit` is invoked.

What happens if an unhandled exception occurs while constructing static objects?
Possibilities:

a. Abort immediately – Simplest to implement, but not consistent with the rule we arrived at after our last round of discussion.

b. Cleanup all fully constructed and partially constructed static objects Possible to implement; however, you could keep getting unhandled exceptions while performing the destruction of static objects. We could keep ploughing on until each object has been destroyed (or not destroyed because an unhandled exception was encountered while destroying it).

>From Margaret Ellis:

If an unhandled exception occurs while constructing static objects, call `terminate`. Are you asking what happens if `terminate` calls `exit`? If `terminate` calls `exit`, any fully constructed or partially constructed statics should be destroyed.

>From Michey Mehta:

What happens if an unhandled exception occurs while destroying static Objects?

Once again, both a. and b. above are possibilities. Note that you could have begun the process of destroying static objects for two reasons:

1. The user program terminated normally.
2. The user program terminated with an unhandled exception.

>From Margaret Ellis:

If an unhandled exception occurs while destroying static objects, call `terminate`. Are you asking what happens if `terminate` calls `exit`? I thought Jonathan answered this in our previous exchange, as follows:

> If `terminate` calls `exit`, try to destroy any remaining static objects. Do not try again to destroy the object that caused the exception.

>From Andy Koenig:

Thoughts on unhandled exceptions, etc.

I think an exception that occurs during construction or destruction of a static object should be treated the same way as any other exception, under the assumption that the constructor/destructor in question was called from an invisible block surrounding `main`.

In other words, the semantics should be similar to the following:

```
{
        construct_statics();
        int ret = main(argc, argv);
        destroy_statics();
}
```

If there's no handler for an exception raised in construct_statics or destroy_statics, the result should

therefore be to call `terminate()`.

>From Jon Shopiro:

[Quoting mail he received as ANSI editor]

> There appears to be an ambiguity in the latest draft ANSI C++ standard regarding what happens when there is a "throw" for which there is no matching handler. I understand that `terminate()` is called. The question is whether the entire stack is unwound, or whether there is no unwinding. In other words, do the destructors of existing auto objects get run, or not, before `terminate()` is called? I could not find anything in the spec that seems to answer this question. Thanks.

We've been having some discussions on this very issue and here is our current thinking. When an unhandled (unexpected) exception is thrown, the function `terminate()` (`unexpected()`) is called. No stack elements or static objects are cleaned up, so if `terminate` enters a debugger (for example) the state of the program is intact.

Things can become more complex (and we're less sure of the answers) depending on what happens next. If `terminate()` calls `exit()`, then `exit` attempts to clean up static objects (automatic objects are still left on the stack). If an exception is thrown and handled during this process the next static object is cleaned up. However if an unhandled exception is thrown during any processing stemming from a call of `terminate()`, the program is immediately aborted.

P.S. I think most programs that care will wrap all processing in a try block with catch clause that catches all exceptions, thus rendering the question moot.

>From dlw:

Editor's Note: I don't know who "dlw" is, but someone forwarded this mail, which seems germane, from him.

We had some experience with these issues in the Symbolics exception system. There, we had an interactive environment, which makes somewhat greater demands on an exception system than the conventional Unix process model, but I think some of the same considerations apply.

I agree that when `terminate()` is called due to an unhandled throw, cleanup should not happen, so that `terminate` can enter a debugger.

However, the program's `terminate()` function might decide that it would be a good idea to run all the cleanups on the stack, before the process terminates. Even though the process is terminating, it might still be useful to run cleanups, because they might affect state outside the program. For example, some object might have a constructor that creates a file, and a destructor that deletes the file just in case there is an unwind. (There would have to be some state bit in the object so that an unwind can delete the file but a normal return can leave it alone; too bad there isn't a way for a destructor to learn whether it's being called due to an unwind or not.) It would be good to be able to make sure such a destructor was called before the program terminates.

Maybe `exit()` should be defined to run cleanups on the stack. Or maybe there should be a function for `terminate()` to call that explicitly runs the cleanups and then returns, so that you can run cleanups and then `abort()`, or something.

5.2 Can `terminate` call `exit`?

5.2.1 RESOLUTION: Yes, `terminate` can call `exit`.

5.2.2 DISCUSSION

>From Michey Mehta:

`terminate` must be called on an unhandled exception, and `terminate` may not return (i.e. it must do something equivalent to `abort`). On page 21 it mentions that abort is instantaneous, and no cleanups take place.

>From Margaret Ellis:

The default action for `terminate` is to call `abort`, but a user-supplied `terminate` function does not have to abort. See ARM 15.6.1:

> An example of a reasonable `terminate()` that does not abort is a function that discards all the data within its address space as probably corrupted and re-initializes the process.

It logically follows that a user-supplied `terminate` function may call `exit`.

>From Michey Mehta:

I suppose it is an error to call `exit` from a `terminate` function (and we could actually have our version of `exit` detect this condition and abort).

>From Margaret Ellis:

That might be a reasonable thing to do, but it involves a change to the language definition.

>From Michey Mehta:

Our feeling is that `terminate` should be very careful about calling `exit`: If you are in `terminate` because an unexpected exception has occurred, then calling `exit` will result in static objects being cleaned up; cleaning up static objects could result in another unexpected exception, which would call `terminate` again which would call `exit` again . . . this could go on ad infinitum . . . the idea of `terminate` being called while a previous `terminate` is still on the stack seems quite dangerous.

I would propose that the language definition be changed to say that if `terminate` calls any code that results in an unhandled exception, then the program will be aborted immediately. Otherwise you are going to have to explain to users that they need to code `terminate` so that it is reentrant, and I don't think this is desirable. My suggestion in a previous message about `terminate` not being permitted to call `exit` does not quite capture what I really want, which is that no unhandled exception should occur after `terminate` is entered.

>From Andy Koenig:

There is one strong advantage to NOT unwinding the stack first, though, and that is that is that a debugging trap planted in `terminate()` will have more information. In other words, if `terminate()` dumps core, the stack will still be there and it will be possible to obtain a traceback from the core dump.

Similarly, it would seem to be useful for `terminate()` to be able to pass control to a debugger, in which case you definitely don't want to unwind the stack first.

>From Margaret Ellis:

I agree. I suspect that's why 15.6 is written the way it is. As I read it, this is what the language def'n calls for.

I think Michey's suggestion that the program abort if an unhandled exception occurs after `terminate` has been called is reasonable. BUT we have to be careful to allow a program to discard its data and re-initialize (per annotation in ARM 15.6.1), and since it is an error for `terminate` to return, one has to be careful about defining what we mean by "after `terminate` has been called." I

mean "after `terminate` has been called but before the process has been re-initialized or some other corrective action has been taken." I'm not sure how to specify this clearly and concisely.

>From Andy Koenig:

It should be a simple matter for the run-time library to check for recursive calls to `terminate()` and call `abort()` in that case.

In practice, I don't think it's a real big deal. I do think, though that the default action for `terminate()` should include destroying statics because otherwise the poor user is likely to miss the last bufferload of output on `cout`.

5.3 Can `unexpected` return?

5.3.1 RESOLUTION: No, `unexpected` can not return.

An attempt to return from `unexpected` is undefined. Our implementation will call `abort()`.

5.3.2 DISCUSSION

>From Andy Koenig:

We need to think about just what `unexpected()` is allowed to do other than abort. If it returns, where does it return to? I agree with Jonathan: `unexpected()` should not be allowed to return.

>From Jon Shopiro:

I think like `terminate()`, it can't return.

>From Margaret Ellis:

Considering that there isn't an obvious reasonable place for `unexpected` to return to, we should disallow returning from `unexpected()`.

>From Jon Shopiro:

When we thought about this it didn't seem clear where `unexpected` should return, but if

```
void foo() throw (X, Y)
{
      /* body */
}
```

is *defined* as

```
void foo() throw (X, Y)
{
      try {
            /* body */
      }
      catch (X) { throw; }
      catch (Y) { throw; }
      catch (...) { unexpected(); }
}
```

`unexpected()` could return in a defined way.

I am opposed to this for two reasons.

1. very surprising semantics
2. you necessarily fall off the end of `foo` which may be expected to return a value.

>From Bjarne Stroustrup:

[`unexpected()` could return in a defined way] Only if we allowed it to. I think we should define `foo() throw (X, Y)` that way and state that `unexpected()` may not return, that an implementation is allowed to give a compile time error if it can detect that an `unexpected()` function returns (hard to do), and it is undefined what happens if `unexpected()` does return.

Suggested implementation:

```
void foo() throw (X, Y)
{
      try {
            /* body */
      }
      catch (X) { throw; }
      catch (Y) { throw; }
      catch (...) { unexpected(); abort(); }
}
```

5.4 Can `unexpected` throw or rethrow?

5.4.1 RESOLUTION: Yes, `unexpected` can throw or rethrow.

5.4.2 DISCUSSION

>From Jon Shopiro:

Recommend disallowing all throw expressions (not just those with no operand) in code dynamically within `unexpected` or `terminate`. You only get to those functions when exception handling is broken.

>From Margaret Ellis:

That's not strictly true. You get to `terminate()` when exception handling is broken, but you can get to `unexpected()` when an unexpected exception is thrown - the exception handling mechanism may still be functioning just hunkey-dorey.

>From Andy Koenig:

Disallowing all throw expressions in code dynamically within `unexpected` or `terminate` is slightly too draconian. The following should surely be legal:

```
void unexpected() {
      try {
            throw E();
      } catch (E) {
            // ...
      }
}
```

>From Jon Shopiro:

This is reasonable, but how important is it? I would definitely advocate prohibiting an exception that propagates through unexpected, and if it made the implementation easier to prohibit all exceptions dynamically within unexpected, I wouldn't miss it. It can always be added later.

>From Andy Koenig:

If `terminate()` is not allowed to call any function that throws an exception, period -- even if that function catches the exception too -- then we will have to say for every function we ever write whether or not it is allowed to use exceptions internally.

>From Andy Koenig:

Here is an example of what I was talking about earlier.

Suppose we have a function like this:

```
extern void f() throw();
extern void g() throw(char*);

main()
{
      try {
            f();
            g();
      } catch (char* msg) {
            // ...
      }
}
```

Looking at this code, I think I have the right to expect that either `f` will return or the entire program will be terminated. In particular, I should be entitled to believe that if I get to the catch clause, I got there because of an exception in `g`.

But what if `f` looks like this?

```
static void fling() { throw "I escaped!"; }

void f() throw()
{
      set_unexpected(fling);
      throw 42;
}
```

If `fling` behaves as if it is called from the context that called `f`, then this trick makes it possible to circumvent the `throw()` clause attached to the definition of `f`, so I can never trust such clauses. In that case, why bother with them at all?

>*From Jon Shopiro:*

Can we define these things in terms of already defined aspects of the language? Then these questions would be easier to answer (although they might have the wrong answers).

Here's a first try...

```
<return_type> f(<args>) throw (<type1>, <type2>)
{
      <body>
}
```

is defined to be

```
<return_type> f(<args>)
{
      try {
            <body>
      }
      catch (<type1>) {
            throw;
      }
      catch (<type2>) {
            throw;
      }
      catch (...) {
                unexpected();  // must be a fn pointer to
                               //   accommodate set_unexpected()
            undefined();   // unexpected may not return
      }
}
```

If this is the definition, it's clear what happens if an exception propagates through `unexpected()`.

Comments?

>From Bjarne Stroustrup:

Except for the `undefined();` (which is undefined :-)) I thought that that WAS the definition.

>From Jon Shopiro:

If exception specifications are defined as in my previous note [then considering Andy's example:]

```
extern void f() throw();
extern void g() throw(char*);

main()
{
      try {
            f();
            g();
      } catch (char* msg) {
            // ...
      }
}
```

if `f()` throws an exception, `unexpected` will be called, which may itself throw an exception. Thus you _could_ get to the catch clause without even entering `g()`.

I won't argue whether this is good or not, but I would just like to get this stuff pinned down to mean _something_.

[Andy says:]

But what if f looks like this?

```
static void fling() { throw "I escaped!"; }

void f() throw()
{
        set_unexpected(fling);
        throw 42;
}
```

If `fling` behaves as if it is called from the context that called `f`, then this trick makes it possible to circumvent the `throw()` clause attached to the definition of `f`, so I can never trust such clauses.

Well, according to the definition I suggested and according to your definition of "trust," you're absolutely right.

[Quoting Andy again:]

> In that case, why bother with them at all?

Well, I can remember arguing against them a couple of years ago, but I lost. :-)

>From Andy Koenig:

Jonathan says:

> if `f()` throws an exception, `unexpected` will be called, which may itself throw an exception. Thus you _could_ get to the catch clause without even entering `g()`.

And my point is that this undesirable state of affairs will never be true if throwing an exception from `unexpected` causes the second exception to appear to originate from the same point as the first.

In other words, in this example:

```
void f() throw()
{
        set_unexpected(fling);
        throw 42;
}
```

I would like the behavior to be the same as:

```
void f() throw()
{
        set_unexpected(fling);
        unexpected();
}
```

which would cause a recursion loop in `unexpected` unless the library detects it and aborts.

>From Jon Shopiro:

I've been doing a little more thinking on exception specifications and the `unexpected()` function.

Andy points out that if exception specifications are interpreted as Bjarne and I suggested, exceptions can be thrown by the `unexpected` handler that violate the exception specification. It is hard to promote this as a feature of the language.

It would be possible to prohibit `unexpected` handlers from throwing exceptions at all. But then

since they can't return they could only shut down the program (without cleaning up the stack), so the guarantee provided by the exception specification would be "no exception outside this list will be thrown (but the program may be shut down)." That would seem to make programs less reliable and robust, not more.

One way to deal with this problem is to revise (as the programmer)

```
int foo(<args>) throw (type1, type2)
{
      <body>
}
```

to

```
class Unexpected { /* no members */ };
int foo(<args>) throw (type1, type2, Unexpected)
{
      try {
            <body>
      }
      catch (type1) { throw; }
      catch (type2) { throw; }
      catch (...) { throw Unexpected(); }
}
```

Then `unexpected()` will never be called, and we don't have to worry about what it does. This transformation could also be built into the language, eliminating the need for `unexpected()`. Alternatively, you could get the same effect by defining

```
void throw_U()
{
      throw Unexpected();
}
```

and calling

```
set_unexpected(throw_U);
```

This seems pretty unsatisfactory to me, since all this does is throw away information (viz., the originally thrown exception) that might be used somewhere. For example, suppose I am writing an I/O library and I put in my functions throw specifications listing the I/O exceptions. But maybe my library does some storage allocation and the storage allocator throws its own exception. Now as the author of the I/O library, I may not know what exception is thrown by the storage allocator (or even what storage allocator my customer is planning to use) so I can't put the storage allocation exception in my exception specification, or I may just feel that storage allocation exceptions are not in my bailiwick and I don't want to put them in my exception specification. It may also be that there is a handler for the storage allocation exception somewhere on the stack that could buy some more core and retry. It doesn't seem very useful for me to catch all unexpected exceptions and turn them into `Unexpected`.

On the other hand, if exception specifications behave the way Andy wants (I hope I'm interpreting him correctly), you could write a program like this:

```
int foo(<args>) throw ()
{
      set_unexpected(throw_U);
      try {
            <body>
      }
      catch (Unexpected) { return 0; }
}
```

Why bother throwing an exception from `unexpected()`? You might as well have written

```
int foo(<args>) throw ()
{
      try {
            <body>
      }
      catch (...) { return 0; }
}
```

If you don't set up your own `unexpected` handler, and the previously existing one throws an exception, it's a pretty sure bet under Andy's interpretation that the program will loop, because the exception the `unexpected` handler throws will be unexpected (practically by definition!).

So, to recapitulate, if `unexpected` handlers are not allowed to throw exceptions, programs become less reliable. If they are allowed to throw exceptions and Bjarne's interpretation is accepted, the exception specifications themselves become suspect. If Andy's interpretation is accepted, programs are likely to loop (leading to an even more uncontrolled crash than `terminate()`) or be unnecessarily obfuscated.

My conclusion is we need a better idea or we don't need exception specifications. Can anybody give an example of a program with exception specifications (any interpretation) that is more reliable than the same program without exception specifications?

>From Margaret Ellis:

Perhaps the mistake we've all been making in trying to figure out what `unexpected` should be allowed to do is that we have been interpreting an exception specification to mean

> I promise that I won't throw anything other than the exceptions in this list.

We've been talking about `unexpected` throwing an exception as a violation of the exception specification. The ARM vacillates on this. In 15.6.2 it talks about "a violation of a promise to throw only a specified set of exceptions," but in that same section it gives an example of an `unexpected` function rethrowing the unexpected exception.

Perhaps the way we should interpret an exception specification is

> I promise that under the right conditions I WILL throw these exceptions.

(See the annotation re: Murphy's Law on 362.)

With this perspective, it no longer follows that allowing `unexpected` to throw or rethrow "circumvents" the exception specification, or "violates" the guarantee of the exception specification.

If we consider an exception specification to be what the function will throw AT LEAST, then we can make use of that information. Reconsider the example Jonathan posed:

> For example, suppose I am writing an I/O library and I put in my functions throw specifications listing the I/O exceptions. But maybe my library does some storage allocation and the storage allocator throws its own exception. Now as the author of the I/O library, I may not know what exception is thrown by the storage allocator (or even what storage allocator my customer is planning to use) so I can't put the storage allocation exception in my exception specification, or I may just feel that storage allocation exceptions are not in my bailiwick and I don't want to put them in my exception specification. It may also be that there is a handler for the storage allocation exception somewhere on the stack that could buy some more core and retry.

Now I can use this library and I know I may have to handle the exceptions that it tells me it will throw. Great. If I also know that my allocator may throw `No_more_memory`, I will write an `unexpected` function that handles `No_more_memory`, and maybe some other exceptions I know my system may throw. If I get to `unexpected` with an exception I'm not prepared to handle, I can rethrow it and maybe somebody deeper in the stack may know what to do with it.

I'm beginning to think exception specifications make sense only if you consider them a promise to throw certain exceptions, and a promise that you get a shot at handling anything else that comes from anything that function may call (in `unexpected`, or in some function that catches what `unexpected` throws), not a promise NOT to throw other exceptions.

>From Bjarne Stroustrup:

```
void f() throw(X,Y);
```

means that `f()` promises its caller only to throw exceptions of type `X` or `Y` and if it tries to throw any other exceptions `unexpected()` will be called.

In principle, the caller choses `unexpected()`. If the caller choses to make unexpected mean `throw Z;` then I think it is entitled to. That might be stupid, but I suspect that many will adopt the Clu convention that `unexpected()` means `throw Fail;`. That is not my favorite convention, but I don't think we should try to outlaw it.

>From Jon Shopiro:

So if the exception specification is a promise that the exceptions on the list will be thrown sometime under some circumstances, but other exceptions might be thrown too, what do we have?

The promise is in some sense meaningless because it cannot be false. The caller can use `set_unexpected` to intercept control when an unexpected exception is about to be received, but the called function can nullify that arrangement, e.g.,

```
void f() throw(X);
void throw_Y() { throw Y(); }
void g()
{
      set_unexpected(throw_Y);
      try {
            f();
      }
      catch (X x) { ... }
      catch (Y y) {
            // f got an unexpected exception
      }
      catch (...) {
            // no way this code can execute (right?)
      }
}
void throw_Z() { throw Z(); }
void f() throw (X)
{
      set_unexpected(throw_Z);  // WRONG!!!
      ...
}
```

My question is, what really useful thing can be done with this feature? It seems the only likely choices for the `unexpected` function are `pass_through` (ARM pg. 365) and terminate. Who needs it?

>From Bjarne Stroustrup:

The default behavior of a function that violates its exception specification is termination (`abort()`). As far as I understand things, that is generally agreed to be acceptable. However, in every discussion we had about exceptions people objected that this wasn't acceptable as the only resolution. I think it was Jonathan who summarized exception specifications like this "Either I obey the rules or else I commit suicide!"

What people wanted was a way for a caller to say "instead of terminating do `X`" where the most popular choices for `X` was "throw Fail" and "record the problem and throw an acceptable exception." I see `unexpected()` as a way for a caller to specify exactly what is meant by catastrophic failure. I see a problem with `unexpected()` returning, but not a problem with `unexpected` throwing an exception.

There is an argument for letting `unexpected()` return and let that cause an undefined value to be returned from the called function. The purpose would be to allow people to let `X` be "set errno and return normally."

Agreed, a function can violate its contract by replacing the callers `unexpected()` with its own, but so what? That is simply very bad programming or fraud.

>From Andy Koenig:

Which is less bad?

1. defining a canonical exception that any function may potentially throw even if it says otherwise, as with the `failure` exception in Clu, or
2. saying that a function may not throw any exception it says it won't throw, but provide a sneak path for users to violate that guarantee whenever they want?

If those are the only two choices, I think (1) is much less bad. However, I still prefer

3. when a function says it will only throw a particular set of exceptions, that's all it will throw, period. A program that tries to use `unexpected()` to cause it to throw something else is just as undefined as one that tries to dereference an uninitialized pointer, and a good implementation should forcibly terminate it.

>From Jon Shopiro:

I think there might be three reasons to give exception specifications:

1. to make programs better documented,
2. as a debugging aid,
3. to make production programs more reliable.

If exception specifications are only for documentation or if they are strictly for debugging, that is, `unexpected()` is never called in production programs, then you might as well use a version of the assert macro. So I assume (and apparently Bjarne agrees) that `unexpected()` is expected to be called occasionally. Bjarne writes:

> The default behavior of a function that violates its exception specification is termination (`abort()`). As far as I understand things, that is generally agreed to be acceptable. However, in every discussion we had about exceptions people objected that this wasn't acceptable as the only resolution. I think it was Jonathan who summarized exception specifications like this "Either I obey the rules or else I commit suicide!"

It seems to me that calling `abort()` is only marginally better than random chaos for a production program. Not worth adding a language feature if that's all it buys you.

> What people wanted was a way for a caller to say "instead of terminating do `X`" where the most popular choices for `X` was "throw Fail"

This at least lets you clean up the stack before terminating. I think when the stack contains objects that represent locked resources this will be essential.

> and "record the problem and throw an acceptable exception."

I think the only acceptable exception is the original one thrown. Otherwise you lose information with no recompense. Also, since unexpected exceptions _are_ expected, what's the point of recording them?

> I see `unexpected()` as a way for a caller to specify exactly what is meant by catastrophic failure.

I guess my main beef is with the phrase "catastrophic failure." I think that `unexpected()` is likely to be called a lot (if people use exception specifications at all) and it better not be a catastrophic failure.

> I see a problem with `unexpected()` returning, but not a problem with `unexpected` throwing an exception.
>
> There is an argument for letting `unexpected()` return and let that cause an undefined value to be returned from the called function. The purpose would be to allow people to let `X` be "set errno and return normally."

You get into worse trouble if the function returns a class that doesn't have a default constructor.

There should be no correct programs that exhibit undefined behavior.

> Agreed, a function can violate its contract by replacing the callers `unexpected()` with its own, but so what? That is simply very bad programming or fraud.

Correct manipulation of `set_unexpected()` is now very tricky. I think you need to set it in a constructor and restore it in a destructor, and you still can't do it in a function with an exception specification.

>From Andy Koenig:

Having gone back to the example in ARM 15.6.2 (page 365) I must say that that example is completely unequivocal. I still think it is the wrong way to go, and would have said so if I had noticed it at the time. However, I didn't, and it's too late to change it now.

This implies that exception specifications can only be considered advisory. It is pretty strong advice, in the sense that the only way not to follow it is to use `set_unexpected`.

5.5 From where does `unexpected` appear to throw?

5.5.1 RESOLUTION: `unexpected` appears to throw (or rethrow) as if it were called from a handler for the try block in which the unexpected exception was raised. That is,

```
<return_type> f(<args>) throw (<type1>, <type2>)
{
        <body>
}
```

is defined to be

```
<return_type> f(<args>)
{
        try {
                <body>
        }
        catch (<type1>) {
                throw;
        }
        catch (<type2>) {
                throw;
        }
        catch (...) {
                unexpected();  // must be a fn pointer to
                               //  accommodate set_unexpected()
                undefined();   // unexpected may not return
        }
}
```

5.5.2 DISCUSSION

>From Jon Shopiro:

I agree that `unexpected()` should not be allowed to return, but I think it should be allowed to throw an exception. The question is where should the exception appear to be thrown from (that is, what catch clauses are eligible to handle the exception and what exception specifications are checked)? In order to guarantee progress, the exception should appear to come from the caller of the function whose exception specification is violated, but this needs more discussion.

>From Bjarne Stroustrup:

Agreed.

(1) pedantry:
The exception is thrown in `unexpected()` so `unexpected` could in principle catch it itself.

(2) `unexpected()` is in principle called by a catch clause implementing an exception specification. Since no code specific to the called function is executed there is no difference between saying that an exception thrown by `unexpected` is thrown by the caller or the callee because the same set of catch clauses will be in effect.

>*From Andy Koenig:*
[Having the exception appear to come from the caller of the function whose exception specification is violated] would mean that calling a function declared with `throw ()` could still appear to throw an exception. I think it would be better if saying `throw ()` means that either the function returns without throwing an exception or it never returns (either by looping or terminating execution altogether). Thus I think that throwing an exception from `unexpected` should appear to come from the throw that caused `unexpected` to be called.

5.6 What does `unexpected` rethrow?

5.6.1 RESOLUTION: A rethrow in `unexpected` rethrows the exception that caused `unexpected` to be called.

5.6.2 DISCUSSION

>*From Michey Mehta:*
`[ . . . ]` you could already be in a handler, and then perform some operation which causes `unexpected` to be called; in this case, a "rethrow" would throw the exception that caused the original handler to be entered (not the exception which caused `unexpected` to be called).

>*From Margaret Ellis:*
Exceptions don't stack, so if you get to `unexpected` from a throw while handling another exception, a rethrow in `unexpected` would have to throw the current exception (the one that landed us in `unexpected`), not the one the other handler was handling.

5.7 Are `terminate` and `unexpected` available without including a header file?

5.7.1 RESOLUTION: No, `terminate` and `unexpected` must be declared. Their declarations will appear in the header `exception.h`.

5.7.2 DISCUSSION

>*From Michey Mehta:*
In test `6/t7.r0` `terminate` is called, but not declared. We used to make this routine available, but now it is only available by including `eh.h`. `Are these routines supposed to be available without including a header file?` **Editor's Note:** Note that the header file is to be named `exception.h`, not `eh.h`.

>*From Margaret Ellis:*
`set_unexpected` and `set_terminate` are not "always there"; the user must include `exception.h` to get them.

This would imply that if `terminate` and `unexpected` are similarly not "always there", their declarations would also belong in `exception.h`.

>*From Andy Koenig:*
I do not think `terminate()` or `unexpected()` should be wired into the compiler. Two reasons:

1. No other functions are: any such things are either reserved words (like `int`), have special syntax (such as operator `new`), or must be explicitly included in appropriate headers (like `size_t`).
2. Users can explicitly define their own `terminate()` or `unexpected()` functions, which will be called by the system. This would be harder if the compiler treated them as magic.

I think `.terminate()` and `unexpected()` should get the same kind of treatment as `set_unexpected()`: library functions that have, and conceal, special knowledge of the run-time system.

>From Bjarne Stroustrup:

I see no reason not to have `terminate()` and `unexpected()` as perfectly ordinary functions.

6. Other Issues

6.1 Can the implementation impose a restriction on aggregates of class types?

6.1.1 RESOLUTION: It will be acceptable for 4.0 to issue a sorry message for an aggregate of a class with a destructor.

6.1.2 DISCUSSION

>From Michey Mehta:

Page 151 of the ARM allows aggregates for classes that:

— have no constructors

— no private or protected members

— no base classes

— no virtual functions

Our EH implementation has added another restriction, which results in the failure of `oca018.c0` in your system test suite:

— no destructor (because we synthesize a ctor if there is a dtor)

Whenever an object with a destructor is created, we need to bump the global `dt_count` since the object will need to be destroyed in the event of an exception. The most convenient way to bump `dt_count` is to have the constructor do this; if there is no constructor, we make one. So we have a case of an implementation convenience that impacts the language accepted by cfront. What is your preference from the choices mentioned below?

1. Have some other mechanism for bumping `dt_count`, instead of using the constructor do it. This will be a lot of work.
2. Change the language definition. No chuckles please.
3. Document this as a bug. Classes that obey the first four requirements on page 151, and also have dtors, should be fairly rare.

>From Margaret Ellis:

Michey says:

3. Document this as a bug.

And issue a sorry, I suppose.

>From Glen McCluskey:

Does this include the case where you have a struct with a destructor?

There is no constructor or private/protected members or base classes or virtuals.

This might be commoner than we think.

>From Jon Shopiro:

Fix or document as bug.

>From Andy Koenig:

I can think of only a single legitimate use for a class with an explicit destructor and no explicit constructor, namely if the destructor is virtual and has a null body. I can't get too worked up over restricting initialization of objects of such classes; I suspect you could get away with saying 'Sorry, not implemented' in this particular case.

6.2 Can the implementation disallow declaring a data member by the same name as its containing class?

6.2.1 RESOLUTION: It will be acceptable for 4.0 to issue a sorry for the appearance of a data member by the same name as the class.

6.2.2 DISCUSSION

>From Michey Mehta:

[...]

Whenever an object with a destructor is created, we need to bump the global `dt_count` since the object will need to be destroyed in the event of an exception. The most convenient way to bump `dt_count` is to have the constructor do this; if there is no constructor, we make one.

[... I noticed an] implication of synthesizing a constructor. The following source is allowed by a non EH cfront, but not by an EH cfront:

```
class C {
   int C; //Bizarre (but legal?) choice of name
   ~C();
};
```

6.3 Are transfers of control into try blocks and handlers legal?

6.3.1 RESOLUTION: No, transfers of control into try blocks and handlers are not legal.

6.3.2 DISCUSSION

>From Jon Shopiro:

I can see where it would make the implementation (even) harder, but it seems semantically well defined so I don't see any reason to prohibit it.

>From Glen McCluskey:

I don't see how this is semantically well defined. I assume that if you `goto` into a try block, for example, then you have NOT changed the global EH state such that you can do a corresponding catch. Is that so?

If you transfer into a catch, then a rethrow should be invalid or should apply only to a catch one level removed. Is that what you had in mind?

The only way I could see to implement this would be to keep a flag in the function and set it just before transferring into the block. Pretty hokey but it could probably be done.

There are other issues, such as:

```
        catch (A a) {
              // stuff
xxx:
              // more stuff
        }
```

If you jump to `xxx`, you may issue a dtor call for a without having done the corresponding copy ctor call when `a` is copied out from the EH runtime storage.

>From Jon Shopiro:

Think of `try` as lexical. Then if you transfer into a try block, subsequent throws can be caught by that try block's handlers. Pretty simple.

A rethrow would be a run time error, just as a rethrow in ordinary code is an error if that code wasn't called from a handler. However, if catch has an argument, then transferring in should be illegal just like jumping into a block with a constructed local.

Allowing transfers into throw and argument-less catch would be more rope, but our users seem to want rope.

>From Bjarne Stroustrup:

We considered `goto`s into try blocks and catch clauses when we wrote the EH paper and decided to ban them. You can probably find a statement to that effect somewhere. Allowing such transfers will cause nothing but confusion.

>From Andy Koenig:

I think jumping into a catch clause is skipping an initialization:

```
try {
      // ...
}
catch (E e) {
      // ...
x:
      // ...
}
```

If you say `goto x;` from outside the catch, you're skipping the initialization of `e`. I think this should hold true even if a formal parameter is not explicitly named; it has to be there anyway.

>From Bjarne Stroustrup:

Indeed. Also, jumping into a try block may also be skipping some initialization because one could imagine implementations where `try` caused some action (reversed by the matching '}').

In general, I think it unwise to complicate things just to allow more `goto`s.

>From Margaret Ellis:

HP already gives an error on attempt to jump into a try block or handler.

6.4 Is it correct to consider an object constructed when its last statement is reached, while a destructor is considered complete just before its first statement is reached?

6.4.1 RESOLUTION: An object is not considered fully constructed until everything in the constructor is finished. An object is considered partially destroyed before anything happens in the destructor.

6.4.2 DISCUSSION

>From Glen McCluskey:

I noticed in looking at EH code that a constructor is considered complete when its last statement is reached (the global count is incremented), while a destructor is considered complete just before its first statement is reached.

Does this seem right?

>From Margaret Ellis:

Not to me.

>From Michey Mehta:

This seems fine to me. An object is not considered fully constructed until everything in the constructor is finished. An object is considered partially destroyed before anything happens in the destructor.

>From Andy Koenig:

Constructing an object involves (recursively) constructing its base class parts and data members and then executing its constructor. While this is going on, it is necessary to remember how much of the object has been constructed because if any part of it throws an exception, it is essential to destroy only what has been constructed.

Exactly the same is true during destruction: to destroy an object, you execute its destructor and then (recursively) destroy its data members and base class parts. While this is going on, it is necessary to remember the state of the destruction so that if any part of it throws an exception, the rest of the object will be destroyed and nothing else.

For example:

```
struct A {
        String p, q;
        A() { foo(); }
        ~A() { bar(); }
};
```

Suppose foo throws an exception when called from the constructor of some `A` object. Then while unwinding the stack, it is essential to destroy members p and q of that object.

Similarly, suppose that while the object is being destroyed, the destructor calls bar, which throws an exception. Then members p and q have not yet been destroyed and it is essential to destroy them while unwinding the stack.

To do anything else would cause completely ordinary exceptions to generate memory leaks that the programmer would have no chance at correcting.

>From Margaret Ellis:

All of which implies that destruction can not be considered complete before the first statement of the dtor.

>From Michey Mehta:

Is there really a problem here? Consider the previous example:

```
struct A {
      String p, q;
      A() { foo(); }
      ~A() { bar(); }
 };
```

We increment dt_values *after* finishing a constructor, and decrement dt_values *before* starting a destructor. Here is the sequence of events during the lifetime of an A object:

```
Begin Constructing A
   Construct String p
     Do String::String stuff
     dt_count=1;
   Construct String q
     Do String::String stuff
     dt_count=2;
   Do A::A stuff
     dt_count=3;

Begin Destroying A
     dt_count=2;
   Do A::~A stuff
   Destroy String q
     dt_count=1;
     Do String::~String stuff
   Destroy String p
     dt_count=0;
     Do String::~String stuff
```

If an exception occurs at any point in the lifetime of this object, the `dt_count` value is sufficient to determine exactly which subobjects need to be destroyed, since our typeinfo information describes the structure of an `A` object. Glen - maybe you were confused because you thought that an increment/decrement of `dt_count` represented construction/destruction of the *entire* object, whereas all it really pertains to is the subobject?

>From Margaret Ellis:

OK. Glen tried to break it and hasn't succeeded - and you know he's pretty good at that!

6.5 Should the EH runtime delete memory allocated by a `new`-with-placement?

6.5.1 RESOLUTION: No, the EH runtime should not delete memory allocated by a `new`-with-placement.

6.5.2 DISCUSSION

>From Michey Mehta:

If an exception occurs during the construction of a heap object, we run destructors on the partially constructed heap object, and then use the appropriate (i.e. class-specific or global) `delete` operator to free up the memory. One of our internal users has suggested that we should never free up memory if it came via a `new` that used placement. This seems quite reasonable to me: can you think of a case when we should delete memory that came via a `new` which used placement?

>From Andy Koenig:

Agreed: if you didn't allocate the memory, you shouldn't free it.

>From Jon Shopiro:

I guess this is the best choice, but it seems rather asymmetrical. This storage will be lost and if that's okay, then why bother to free any storage when an exception occurs while constructing a heap object?

My view is that this is a symptom of the messiness of the storage management aspect of the language, not a fundamental problem.

>From Andy Koenig:

Maybe I'm missing something.

The most common use of `new`-with-placement is something like

```
p = get_me_some_memory_please();
new(p) T[37];
```

If an exception occurs while constructing one of the `T` objects, the exception handler should destroy the ones that were constructed so far, but it has no way of knowing what to do about `p`.

>From Jon Shopiro:

That's the point. Half of the `new/delete` stuff is well controlled and contained in the language and the other half is an uncontrolled access path to whatever the programmer wants to do. Just for example, note the lack of any placement `delete`. Also, the standard operator `new()`

```
#include <new.h>

void* operator new (size_t, void* p)
{
      return p;
}
```

which Andy's example implicitly uses, is a hook that allows placement `new` to be used in two radically different ways, with a class-internal operator `new()` or with an external memory allocation function as above. The language doesn't distinguish these uses as would be necessary to give a satisfactory answer to Michey's question.

6.6 Should locals and globals be cleaned up when an unhandleable exception is thrown?

6.6.1 RESOLUTION: No, locals and globals are not to be cleaned up when an unhandleable exception is thrown.

6.6.2 DISCUSSION

>From Glen McCluskey:

I can't find it in the standard, but the current implementation does not clean up either locals or globals if an unhandleable exception is thrown.

This makes sense but might be worth clarifying.

```
#include <stdio.h>

struct A {
      A() {puts("A::A"); fflush(stdout);}
      ~A() {puts("A::~A"); fflush(stdout);}
};

void f()
{
      A a;
      throw -97;
}
```

```
main()
{
        try {
                f();
        }
        catch (double) {
        }
}
```

>From Andy Koenig:

I can see some strong arguments IN FAVOR of cleaning up globals in the default implementation of `unexpected()`. Suppose, for example, that global cleanup is what causes output buffers to be flushed. Then it would almost surely be an advantage for all the output to be written before the program ends.

I think `terminate()` is a different story, though -- one expects `terminate()` to be called only if something is drastically wrong, in which case cleanup efforts are likely to make it harder to find the problem.

>From Margaret Ellis:

The annotation in 15.3 reads:

> If a debugger is active for a running program and the program throws an exception that is not caught, it would be unfortunate if the stack were unwound before entry into the debugger.

15.6.2 says

> The default function called by unexpected() is terminate().

I think this implies that locals and globals are not to be cleaned up when an unhandleable exception is thrown.

>From Glen McCluskey:

If `terminate()` is changed using `set_terminate()` and the function calls `exit()`, then dtors for globals WILL be called.

6.7 Should an object for which a destructor has been called still be cleaned up by the EH runtime?

6.7.1 RESOLUTION: A destructor should not be called explicitly on an object for which a destructor will be called implicitly. Thus the EH runtime should not have to worry about whether an explicit destructor call has been issued for an object.

6.7.2 DISCUSSION

>From Michey Mehta:

The ARM allows a programmer to explicitly invoke destructors for an object. When an exception occurs, we clean up all objects that we consider alive, even though they may have been explicitly destroyed before. Is this a problem? Perhaps the ARM should explicitly state that if a user explicitly invokes a destructor on an auto object, the destructor should record this fact in the object, and recognize this if it is ever reinvoked on the same object.

>From Margaret Ellis:

The case you present is analogous to:

```
extern "C" {void printf(const char*, ... );}
class C{
public:
      C() {printf("C's ctor called0);}
      ~C() {printf("C's dtor called0);}
};

C c1;

void main()
{
      C c2;
      c1.~C();
      c2.~C();
      return;  // dtor will be called implicitly for c1 and c2
}
```

I compiled and executed (using cfront 3.1) and I get:

```
C's ctor called
C's ctor called
C's dtor called
C's dtor called
C's dtor called
C's dtor called
```

So with cfront, at any rate, the dtor will be called multiple times for an automatic or statically constructed object if the user has explicitly called a destructor.

It is obviously wrong to allow an explicit invocation of a destructor on an object for which a destructor will be called implicitly. Jonathan says he will change the ANSI draft standard to that effect. There is a remote possibility that the committee will object, but I doubt it.

>From Jon Shopiro:

At end of section 12.4:

> A destructor may not be invoked more than once for an object, nor for may a destructor be invoked for a non-object (except 0). For example, explicitly invoking a destructor on an automatic object would render illegal the implicit destruction that occurs when leaving the scope in which the object is declared.

6.8 How should the EH runtime allocate memory?

6.8.1 RESOLUTION: The implementation will allocate static buffers from which to the RTS will use memory as needed. If the RTS runs out of memory (which could happen only if multiple exceptions were active, or if a very large object is thrown), it will request more memory from `malloc` (which will not be returned). If the request to `malloc` fails, the RTS will abort.

6.8.2 DISCUSSION

>From Glen McCluskey:

Below is a program that simulates the case where you write your own operator `new()` and it throws an exception when it runs out of space.

The result is quite painful; much EH runtime churning and eventually a crash.

I think the EH runtime should use `malloc()` to get rid of this problem. We will need to characterize system-level usage as it pertains to EH, and this is one of the issues.

```
// check EH library allocation

extern "C" {
      char* malloc(unsigned int);
      void printf(const char*, ...);
};

const int SpaceErr = 97;

void* operator new(unsigned int n)
{
      static int i = 0;
      printf("%d0, n);
      if (++i >= 3)
            throw SpaceErr;
      return malloc(n);
}

struct A {
      char x[54321];
};

void f()
{
      A a;

      throw a;
}

main()
{
      int x;

      for (;;) {
            try {
                  f();
            }
            catch (A a) {
                  x = 47;
            }
      }
}
```

>From Paul Faust:

There is a routine called `set_eh_new` that is analogous to the `set_terminate` and `set_unexpected` functions. If you overload system new such that it will execute a throw, you need to set this function to a routine that will not throw.

The other alternative would be to do something as you proposed. However, if there is truly an out of space error, `malloc` would return 0 and the best the run time could do would be to abort.

With `set_eh_new`, the user could stash away just enough space to allow the run-time to continue execution to an appropriate catch point. At that time, the application could chose to reclaim some memory and continue execution.

>From Glen McCluskey:

I don't know if I consider this a real bug, but I would like to request it (change to `malloc`).

Aborting is better than getting into a loop and other sorts of flakiness.

It might also be worth doing other kinds of bullet proofing. For example, in a few places the runtime does divide (/). If you do a divide by 0 and there is a standard exception hooked up to `SIGFPE`, then the result will be very unpleasant. It would perhaps be better to trigger an `assert/abort` in such a case.

>From Paul Faust:

Let me try another path...

C++ has tried very hard to allow the user various mechanisms for the allocation of memory:

1. Default global `new`.
2. User defined global `new`.
3. New for a class of objects.
4. Placement `new` on a particular allocation of an object.

We feel that this model needs to be extended for exceptions. The allocation of memory during the processing of an exception is not the allocation that is done during normal execution of the program. Hence the need for `set_eh_new`.

In your example, you illustrate a perfectly reasonable global `new`. In the event of a `malloc` failure, an exception is thrown. If the run-time was modified to call `malloc` when doing the copy, the run-time would hit the same error as the user's code, and the only recourse would be to call `terminate`. Instead, the user could define a routine that would keep static memory in reserve that could accommodate the size of the exception thrown. The user would make this known to the run-time via `set_eh_new`. When the exception occurred, the copy could be made and processing could continue to a handler that knew how to recover from `malloc` problems and would allow the application to continue.

Additionally, users overload new to prevent `malloc` from being called in favor of their own memory allocation system. Calling `malloc` directly would circumvent this.

Lastly, `new` may have caused an exception to be thrown in the processing of application, however, it cannot be assumed that reinvoking the function will cause a new exception to be thrown.

Basically, I think it wouldn't be a good idea to call `malloc` directly from the run-time. It should be emphasized in the documentation that the problem you illustrated can happen, and should be no different than other conditions that lead to infinite loops.

>From Glen McCluskey:

You [Paul Faust] make some very good points about observing new handlers, customers who supply their own `new`, and so on.

However, some group of users will lose out either way. If we use `malloc()`, then customers cannot supply their own storage management, as you say.

If we do support calling operator `new` from the EH runtime, then any `new` called this way cannot throw an exception to globally signal that the application is out of space. Such an exception would result in nested throw processing, with ugly results.

>From a systems programming perspective, which is what the EH runtime is, I consider how it gets its space not to be a user-level issue.

The alternative is to restrict the function of operator `new` so that race conditions do not come up.

>From Michey Mehta:

Some comments on this topic:

— When we did our original design, we wanted the memory allocation to be fast. In retrospect, given the number of things that need to be done when a throw occurs, the cost of the memory allocation is trivial... `malloc` would be reasonable.

— Our "fast" memory allocator assumed that memory allocation follows a stack discipline. However, this is not true in the presence of threads, so our design needs to be modified anyway.

— The library committee of ANSI C++ is currently making the throwing of an exception the *default* behavior of `new` when it cannot get more memory. There is controversy about this. In any case, using such a new in the runtime is problematic.

— Calling `new` or `malloc` should be a last resort. We should always have a static buffer available from which we carve up memory . . . only if this gets empty, should we try to get memory elsewhere.

Conclusion:
The RTS should get out of the business of having yet another memory management algorithm. The ideal solution would be to have a static buffer (let's say 8K) and specify this arena to `malloc(3X)` which would manage it. If this gets exhausted, then use `malloc` to get more memory from the system. But, I think `malloc(3X)` where an arena can be specified is not available everywhere . . . is there any other portable way of managing a chunk of memory?

>From Jon Shopiro:

My view is that EH is invoked when things are going wrong in a C++ program, and so it must be the most robust part of the system. Only `abort()` needs to be more bulletproof (because `abort()` is called when EH fails). Therefore EH should avoid doing any operations that may fail. In particular it should not call `malloc()` and if there is a capability for users to replace its storage management scheme, it should be discouraged, e.g., documented in Greek, or requiring rebuilding of the EH runtime rather than linking in a new module at program loadtime (I'm serious about the latter). If anything in exception processing fails, call terminate(), and if anything in terminate fails, call `abort()`.

Unfortunately in these days of large processes, it's common for `abort()` to fail. Oh well.

>From Margaret Ellis:

I think Michey is right: `malloc` that allows the user to specify an arena is not available everywhere. My documentation for `malloc(3X)` says nothing about managing multiple arenas.

How about these possibilities for portably managing memory?

— Grab a chunk at start-up; when an exception is thrown, you can free your (reserved) space and then proceed to call `malloc` and expect to get it back in the size pieces you need. (This can fail in the face of multiple threads of execution.) If execution is to continue after the exception is handled, the EH RTS will need to again reserve some space - perhaps by snarfing it before returning control back to the application.

— Reserve space statically; the EH RTS will have to manage it for itself. Essentially, this is Paul Faust's idea except that it relieves the user of the burden for reserving the memory.

Again, the RTS has to re-reserve before returning control to the application.

I agree with Jonathan that EH has to be robust. I'd also like to see the responsibility for having enough space to throw/handle an exception fall on the EH RTS, not the user.

>From Michey Mehta:

We have changed our implementation to allocate static buffers from which we carve up memory as needed (we manage the memory ourselves). In the unlikely event that we run out of memory (this would involve multiple active exceptions or very large exception objects) we go begging for more memory from `malloc` (we never returns this memory). If it fails to give us memory, we abort.

6.9 Should `new` throw an exception when it fails?

6.9.1 RESOLUTION: No, `new` should not throw an exception when it fails.

6.9.2 DISCUSSION

>From Michey Mehta:

The library committee of ANSI C++ is currently making the throwing of an exception the *default* behavior of `new` when it cannot get more memory. There is controversy about this. In any case, using such a new in the runtime is problematic.

>From Glen McCluskey:

I think we should agitate against [making the throwing of an exception the *default* behavior of `new` when it cannot get more memory]. If you don't want exception baggage and are content to test the return value against 0, who is to say you are wrong?

>From Margaret Ellis:

I would like to add the observation that there would be a compatibility problem for programs that lock and free resources by a mechanism other than ctors and dtors if `new` throws an exception (the general case of this incompatibility was pointed out previously by Jonathan). We do not want our implementation to force significant recoding on our users, so we should not throw from `new`.

If ANSI eventually standardizes throwing from `new`, we'll implement it then (and probably support two versions for an interval).

Meanwhile, I propose to close this issue by our agreeing that we will not throw an exception when `new` fails (this is consistent with what HP has implemented to date). Copacetic?

>From Jon Shopiro:

I agree with Peggy, but I think we should also make it easy for users to get the other behavior by either supplying an alternative global operator `new` or by supplying a `new` handler function that throws an exception. These should be supplied as source code that can be compiled and linked to an application in the usual way. We should test these as thoroughly as we test the rest of our system.

6.10 Can user programs use `setjmp/longjmp`?

6.10.1 RESOLUTION: The implementation should provide the following minimal level of integration/compatibility with `setjmp/longjmp`:

1. Use of `setjmp/longjmp` w/ EH should not dump core.
2. `setjmp/longjmp` should work correctly when the user program does not use exceptions.

Further integration can wait until ANSI addresses the matter.

6.10.2 DISCUSSION

>From Glen McCluskey:

I assume that with the advent of EH, people who explicitly use `setjmp/longjmp` for any reason are out in the cold, even if they don't care about cleaning up locals.

I haven't tried this but I would expect that it would hose the global list of activation records kept by the EH runtimes. You would bail out of a function without the runtime system knowing it.

Here is an example that crashes. Note that `f()` and `g()` have nothing to do with EH (`try/throw/catch`).

Presumably the crash occurs because the bailout from `g()` leaves around an invalid EH activation record state.

I don't see an obvious solution to this problem save to warn users.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf jb;

struct A {
      int x;
      A(int i) {x = i;}
      ~A() {printf("A::~A[%d]0, x);}
};

void g()
{
      A a(1);

      longjmp(jb, 1);
}

void f()
{
      int x;
      if (setjmp(jb) == 0)
            g();
      x = 37;
}

main()
{
      try {
            f();
            printf("about to throw 970);
            throw 97;
      }
      catch (int i) {
            printf("caught int0);
      }
}
```

>*From Andy Koenig:*

One alternative might be to define C++ versions of `setjmp` and `longjmp` that have hooks into the EH stack unwinding mechanism. To be truly complete, those would have to replace the C versions as well in mixed C/C++ environments.

>*From Glen McCluskey:*

This strikes me as overkill in practice. We would have to implement nonportable versions of `setjmp/longjmp` for every architecture type. For example, on Suns that means Sun-3, 386, and SPARC.

For every architecture type we missed, behavior of programs would be different than for architectures we covered.

>From Jon Shopiro:

There seem to be several degrees of integration of `setjmp/longjmp` with C++/EH. The minimum requirement is that it shouldn't crash, that you should be able to longjmp in code that doesn't use EH at all. Higher levels include being able to use `setjmp/longjmp` in a program that uses EH, but not (dynamically) in a try block, then in a try block, then be able to `longjmp` across try blocks, etc. The final step is that `longjmp` should unwind the stack, calling destructors as it goes.

I suspect the committee will feel that since the stack unwinding mechanism is there for EH, it should be required for `setjmp/longjmp` too.

>From Andy Koenig:

Maybe so, but the library WG was pretty well dead set against it at the London meeting.

>From Glen McCluskey:

One hack that I have tried would be to write a C program like that shown below. This defines a couple of functions `setjmp2()` and `longjmp2()` which are used to save the global EH marker and restore it.

This makes the original example work correctly [see above].

At the minimum we would have to work out the details of packaging this so that these functions are wrapped together with the normal `setjmp()` and `longjmp()`.

There are also other issues. For example, should an explicit `longjmp()` past a bunch of stack frames result in object cleanup for those frames?

```
typedef struct __eh_marker T;

extern T* head__11__eh_marker;

void setjmp2(p)
T** p;
{
      *p = head__11__eh_marker;
}

void longjmp2(p)
T** p;
{
      head__11__eh_marker = *p;
}
```

>From Margaret Ellis:

For our release, I would like to see the implementation at least satisfy Jonathan's two lowest levels of integration:

1. use of `setjmp/longjmp` w/ EH should not dump core.
2. `setjmp/longjmp` should work correctly when the user program does not use exceptions.

>From Glen McCluskey:

POSIX defines two new functions `sigsetjmp()` and `siglongjmp()` that are supposed to be used when bailing out of a signal handler.

This needs to be tracked when fixing `setjmp()/longjmp()` to work with EH.

6.11 Should `exit` throw a standard exception to ensure that automatics are cleaned up?

6.11.1 RESOLUTION: No, `exit` should not throw an exception.

6.11.2 DISCUSSION

>From Andy Koenig:

Jonathan and others have suggested that `exit` should be reimplemented to throw an exception in order to ensure that automatics are cleaned up properly.

I just discovered a pitfall in that approach:

```
void f() { exit(0); }
```

The author of `f` expected it to terminate the program, right? Well...

```
main()
{
        try {
                f();
        } catch(...) {
                // ...
        }
}
```

If `exit` throws an exception, the `catch(...)` will catch it! --unless we say something like "`exit` throws a magic exception that will not be caught even by `catch(...)`."

That's OK, I think -- but we have to remember to do it.

>From Bjarne Stroustrup:

Please notice that it is very common for people to use

```
exit(val)
```

instead of

```
return val
```

in `main()`.

Defining `exit()` in terms of `throw` would make the majority of programs on Suns etc. throw exceptions.

>From Jon Shopiro:

Andy is worried that if `exit()` throws an exception, then `catch(...)` will catch it. I think that's just what you would like to have happen. The purpose of a `catch(...)` handler is to trap surprises. At that point you clean up as best you can and exit. Expanding Andy's program skeleton a little, we get:

```
main()
{
        try {
                f();
        } catch(...) {
                // clean up
                exit(0);
        }
}
```

Now this loses the argument to the first call of `exit`, but that could be dealt with by supplying a catch clause for the exit exception, if you like. The serious objection is that it changes the behavior of existing programs and of course that matters. I think a reasonable approach would be to define a replacement for `exit` (`Exit`?) and let new programs use that but leave the old `exit` and old programs alone. I think if `Exit` that threw a standard exception existed, most programmers would prefer to use it.

Andy suggests as an alternative that we define a special exception that will not be caught by `catch(...)`. That seems an unnecessary complication.

Bjarne comments that this would make most programs throw exceptions. I agree. But I can't tell whether he meant this as an argument for or against the idea.

>From Andy Koenig:

If `exit()` did not exist, we would have to invent it. :-)

Until exceptions, there was a reliable way of forcibly terminating the entire program, namely to call `exit`. If the advent of exceptions takes this away, and we do not provide another standard way of doing it, implementors will inevitably invent their own.

Given that, why not stick with the present meaning? If you want to stop the program, you call `exit`. If you want to throw an exception, throw it.

6.12 What should happen when an exception is thrown from a function registered with `atexit()`?

6.12.1 RESOLUTION: When an exception is thrown from a function registered with `atexit()` `terminate()` should be called.

6.12.2 DISCUSSION

>From Glen McCluskey:

Suppose an exception is thrown from a function registered with `atexit()`? Should this work?

These functions are called during `exit()` processing. But so are static destructors, and the order may matter.

>From Paul Faust:

If the exception propagates past the current function, I believe the correct behavior would be to clean up the locals of the function and then follow the reversed `__sti_list` and call the appropriate destructors.

>From Jon Shopiro:

I think if a function called during exit processing, either a function registered with `atexit()` or a destructor of a static object, tries to exit using an exception, the proper behavior is to call `terminate()`.

6.13 What should happen if the user program calls `alloca()`?

6.13.1 RESOLUTION: If it an be done with reasonable effort, we will support `alloca()` on systems where it is available.

6.13.2 DISCUSSION

>From Glen McCluskey:

`alloca()` is used to increase the stack frame size to get "dynamic" space, which is reclaimed when the stack frame goes away (the function exits).

This has a connection with EH because in the stack unwinding version they need to know offsets into the stack frame for purposes of unwinding.

HP said that they don't support `alloca()` on platforms of interest, so there is no issue for them (but there might be for us).

>From Jon Shopiro:

I'm pretty sure `alloca()` isn't in ANSI C and I hope we don't have to support it. It will make life a lot more complicated for everyone.

>From Margaret Ellis:

It is not [in ANSI C].

I think we can say that if you use `alloca`, then you throw an exception at your own risk.

>From Jon Shopiro:

My view (as I have said before) is that exceptions must be the most robust part of the system. If `alloca()` is included in the library, exceptions must work with it. Sorry.

>From Margaret Ellis:

Glen points out that section 6.13 of the first issue of the EH issues doc contradicts itself.

What is the story on `alloca`?

— My System V manuals do not mention it.

— 9th Edition Unix manuals do not mention it.

— HP does not support it.

Is it standard in BSD? Or is it something only available on SPARC?

I said earlier "I see that we have supported `alloca` in our releases to date." Actually, all I see is the file `alloca.h` in `incl-master/incl`. This header contains (in its entirety):

```
/*ident "@(#)cls4:incl-master/proto-headers/alloca.h    1.1" */
#ifndef __ALLOCA_H
#define __ALLOCA_H

/*      @(#)alloca.h    1.3     88/02/07        SMI     */
#if defined(sparc)
# define alloca(x) __builtin_alloca(x)
#endif


extern "C" {
        char *__builtin_alloca(int);
}
#endif
```

I'm no longer certain what we should be doing with this beast.

>From Glen McCluskey:

The new and excellent book *Advanced Programming in the UNIX Environment* says this:

> The disadvantage is that some systems can't support `alloca`, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

I would say that we should try to support it if we can, but it's not worth working too hard over.

>From Jon Shopiro:

I wrote:

> My view (as I have said before) is that exceptions must be the most robust part of the system. If `alloca()` is included in the library, exceptions must work with it. Sorry.

What I should have added is:

> Therefore `alloca()` should not be included in the library.

7. Acknowledgements

Thanks to all who participated in the discussion that lead to resolutions of these issues.

Special thanks to Glen McClusky, who spotted several errors in the first issues of this paper and suggested several clarifications.

And then, of course, there's Bjarne. If it hadn't been for him, we'd all be working on some other language.