Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS*

Doc No: X3J16/90-0045
Date: July 6, 1990
Project: C++ Extensions WG
Ref Doc: X3J16/90-0042
Reply To: William M. Miller

Position Paper

on

RESUMPTION IN EXCEPTION HANDLING

Since I've been stirring the pot regarding resumption in exception handling for over two years (I wrote an article advocating it in the May, 1988 issue of Computer Language magazine), I feel constrained to contribute my $0.02 worth.

I basically agree strongly with the position put forth by Martin O'Riordan in his email x3j16-ext-29, except that my proposal is a little less extensive than his. Judging from what he wrote there, he wants to extend the scheme presented in the most recent Koenig and Stroustrup paper (X3J16/90-0042, henceforth "K&S") to allow for resumable and non-resumable throw clauses. I would be content merely to provide a `resumable_exception` class as part of the standard library. The net result would be approximately the same. I'll put forward a specific proposal at the end of this document, but first I want to deal with some of the objections to resumption and related issues put forth in K&S and in Bjarne's email touching on the subject, x3j16-ext-20.

In that letter, Bjarne calls resumption "a complicating extension that in my opinion has no value and violates the fundamental notion of what I'm trying to do." With no disrespect intended toward Bjarne, the issue of the value of resumption and whether his goals are the correct ones is open to debate.

Whether resumption indeed "has no value" should be decided by the objective criterion of whether there are useful applications of the feature, not on the basis of opinion. I claim that there is already an existence proof of the usefulness of resumption in the current definition of C++, namely the `set_new_handler()` functionality. The `new_handler` function is called when an allocation fails because the heap is full; if the `new_handler` function returns, the allocation is retried on the assumption that some memory may have been freed. This is resumption, pure and simple! Since section 12 of K&S explicitly lists "memory exhaustion" as one of the intended standard exceptions, resumption clearly has value in this context as demonstrated by its inclusion in the current definition.

---

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

As another example, consider a personal computer application that needs to access a floppy disk. If no diskette is present in the drive, the user must be prompted to rectify the situation, or perhaps the entire program needs to be aborted. This is exactly the kind of situation for which exceptions are intended, since the floppy driver code which discovers the problem has no way of determining how to prompt the user: is it a GUI application, in character mode, or even running over a communications link? Should the prompt be in English, French, or Japanese? Etc.

It's apparent that the decision of how to handle the problem should be made by the calling application; if there are a number of places in the program that do floppy I/O, it will be most convenient to attach the handler to a central location, i.e., one that will be on the stack regardless of which call path discovers the drive to be empty. However, it is obviously impractical to discard the entire calling context from the central dispatcher down to the caller of the floppy driver, as would be required by the termination model. Resumption provides an ideal solution to the dilemma -- the handler initiates a dialogue with the user, and if the user indicates that the fault has been resolved, the handler simply returns to the driver that threw the exception and the access is retried. The alternative without resumption is either massive redundancy, forcing each caller of the floppy driver to have its own handler, or abandonment of exception handling altogether in favor of a less convenient error handling methodology.

If the K&S proposal is extended to include asynchronous events -- an idea I will address further below -- the resumption model becomes not merely convenient but a logical necessity. I am not privy to the detailed inner workings of the Microsoft software mentioned by Martin O'Riordan in x3j16-ext-29, but he indicates that there is a substantial need for resumption in their code, as well. My conclusion is that, far from "having no value," there are a number of contexts in which resumption is eminently useful.

Bjarne continues in x3j16-ext-20, "Adding multiple entry points for functions would be a logically similar operation to adding resumption." Obviously, I disagree with that analysis. Instead, I would argue that forbidding resumption is more akin to introducing multiple inheritance without virtual base classes. Virtual base classes clearly complicate both the definition and the implementation of inheritance, and there is a large set of applications which can get along very well without them. Nevertheless, they were included in spite of these considerations, presumably because the set of applications in which multiple inheritance could be used is larger as a result.

The parallels are strong. The arguments against resumption are similar: K&S section 7 talks about complications, x3j16-ext-20 talks about "diluting a clean model of what the EH mechanism does." Yet, as demonstrated earlier, the set of applications addressed directly by exception handling would be measurably expanded by including (or rather, by not precluding) resumption.

Turning now to the K&S paper itself, the first objection raised is found in section 7:

> if an exception handler can return, that means that a program that throws an exception must assume that it will get control back. Thus, in a context like this:
>
> ```
> if (something_went_wrong) throw zxc();
> ```

> it would not be possible to be assured that something_went_wrong is false in the code following the test because the zxc handler might resume from the point of the exception... With resumption possible, throwing an exception ceases to be a reliable way of escaping from a context.

This idea is presumably the grounds for Bjarne's assertion that resumption "violates the fundamental notion of what I'm trying to do." Frankly, I'm very surprised that this argument is still found in the current version of the K&S paper. As far as I know, no one in the resumption camp advocates making every throw of every exception resumable. I've certainly stated my position on that issue numerous times, both with Bjarne physically present (at the USENIX C++ Conference Advanced Topics Workshop) and in online conferences he reads regularly (on BIX). In fact, the code in my paper (cited as reference 11 in K&S) provides the capability of determining on a per-exception and/or a per-throw basis whether resumption is permitted. Lest there be any confusion, my proposal DOES NOT ALLOW FOR RESUMPTION FROM A PLAIN VANILLA THROW; it is as "reliable" as in the K&S version. This argument is a straw man.

The paper continues:

> Exception handling implies termination

This appears to be nothing more than assuming the conclusion, an attempt to define the opposing position out of existence. It is evident that others have a broader definition of the term "exception handling," and there is nothing in the paper or elsewhere that I've seen to attack the legitimacy of a more inclusive understanding.

Next comes the first of two suggested alternatives to resumption:

> resumption can be achieved through ordinary function calls. For example:
>
> ```
> void problem_X_handler(arguments)       // pseudo code
> {
>     // ...
>     if (we_cannot_recover) throw X("Oops!");
> }
> ```
>
> Here, a function simulates an exception that may or may not resume. We have our doubts about the wisdom of using *any* strategy that relies on conditional resumption...

Is this statement to be interpreted as second thoughts about the new_handler functionality?

> ... but it is achievable through ordinary language mechanisms

While true, this argument does not seem to be terribly persuasive in light of the entire exception handling proposal. In fact, everything provided by the K&S proposal is "achievable through ordinary language mechanisms," except for invoking the destructors of automatic objects in discarded stack frames. If, in fact, the spirit of minimalism is to be the deciding factor on resumption, we should ask whether such an elaborate proposal as K&S is warranted; perhaps instead we should simply require that C++ implementations of longjmp() be defined to do the requisite cleanup activities and leave the rest to "ordinary language mechanisms?" (To paraphrase the last sentence of section

13 of K&S, throwing an exception is "simply an obscure way of spelling `longjmp()`." :-)

I'm not really advocating that we do away with exception handling and just rely on `longjmp()`, of course; my contention, though, is that the reasons that make an ambitious exception handling specification desirable are also applicable to the inclusion of resumption in that specification. What is gained by adding exception handling to the language instead of leaving it to implementation by "ordinary language mechanisms?" To my mind, the premier advantage is that *every* library will be expected to use the same error handling facility. The benefits listed in section 13 of K&S really only accrue on the basis of widespread, indeed, near-universal use of the mechanism provided, and that popularity depends on a) the guarantee that the facility will be present, i.e., part of the language, and b) the fact that it's easy to use. We've already seen that there are a number of applications for resumption -- why should it be an orphan stepchild, subject to the same sort of roll-your-own fragmentation that the K&S approach alleviates for the termination case?

The K&S suggestion, that is, to rely on auxiliary functions to provide resumability, is a major complication over incorporating a resumption capability directly into the exception handling scheme. Consider the number of entities (names or independent pieces of code) involved in implementing resumption in the two approaches: in the case of a built-in resumption capability, there are only two, the specific exception being thrown and a handler block for it. Using the suggested substitute, even assuming a management facility like that described in K&S Appendix G, there are at least four: the exception, the handler, the auxiliary function, and the stacking class object. Furthermore, unlike the built-in case, there is no organic connection among the entities: they are separable. The stacking class object is not necessarily coterminous with the "try" block, and the auxiliary function is, of necessity, completely disjoint from the handler block which will be invoked if the auxiliary function decides to throw an exception rather than resume. The potential exists for mismatch among the pieces; no such mismatch is possible if the handler block itself contains the code to decide whether resumption is possible or not.

To summarize, relegating resumption to "ordinary language mechanisms" while providing a full-blown termination facility is inconsistent and gives unnecessarily short shrift to applications requiring resumption, perpetuating idiosyncratic error handling and giving rise to potentially buggy client code.

The next argument raised in K&S against resumption describes possible bugs introduced by resumption:

> Consider, for example, an exception handler trying to correct an error condition by changing some state variable. Code executed in the function call chain that led from the block with the handler to the function that threw the exception might have made decisions that depended on that state variable. This would leave the program in a state that was impossible without exception handling; that is, we would have introduced a brand new kind of bug that is very nasty and hard to find.

While it is true that resumption, like nearly any new feature added to a language, will allow new and improved bugs as well as new and improved correct processing, the scenario described is not very plausible. Many resumable exceptions will deal with

problems in the program's environment, such as the heap or a floppy disk drive, and not with variables and objects directly visible to the program. Repair of these environmental conditions should have no impact at all in already-executed code. Most other resumable exceptions will have a narrowly-defined interface between the handler and the throw point: the exception itself will carry the data about the problem and the proposed solution. Using some global state variable as the interface is a poor programming practice and unlikely to occur in code written by anyone who knows what he/she's doing. The general rule in the design of C++ has been to give knowledgeable programmers power to do their job better with reasonable safety against accidents, not to try to prevent inept programmers from making a mess of things. Resumption falls squarely in the middle of this tradition.

K&S continues with another suggested alternative to resumption:

> In general, it is much safer to re-try the operation that failed from the exception handler than to resume the operation at the point where the exception was thrown.

This argument would have more weight -- indeed, I think it would be decisive -- in a single-level propagation scheme like those described by Mike Tiemann (G++) and Rob Seliger (Extended-C++). However, it is a fundamental assumption of the K&S model that "there are often several levels of function calls between the point of error and a caller that knows enough to handle the error" (section 13). Consequently, retrying from the handler is potentially both expensive and error-prone. The expense is obvious: quite a great deal of processing may have occurred in the discarded functions that must be repeated in retrying from the handler. The possibility of bugs arises because of potential duplication of actions that should not be repeated. Clearly, if interaction with the user occurs in that code, the integrity of the user interface will be violated by the presence of redundant prompts or notifications. Other bugs may result from the unintentional repetition of initialization or serialization code. In sum, retrying will be more expensive than resumption and not demonstrably less buggy.

The next objection is based on complexity:

> If it were possible to resume execution from an exception handler, that would force the unwinding of the stack to be deferred until the exception handler exits. If the handler had access to the local variables of its surrounding scope without unwinding the stack first we would have introduced an equivalent to nested functions. This would complicate either the implementation/semantics or the writing of handlers.

While it is true that deferring stack unwinding until after the handler has completed its execution will indeed add to the complexity of the implementation, the problems are neither novel nor insoluble, having been dealt with successfully in numerous other languages. I have designed a simple, portable approach to allow handlers access to local variables in a C-generating implementation like **cfront**, and Andy Koenig has agreed that it was feasible. (If anyone is interested in details, please ask; this document is long enough already that I don't want to include it just to show that it can be done.) The real issue is not the added complexity, which is relatively minor, but whether the complexity is worth it. On that count I have little to say other than to refer to the arguments I've already made: that there are real applications for resumption, that the alternatives are unpalatable, and that resumption ought to be

judged on the same basis as the other features of the exception handling scheme: new language syntax, runtime type representation, and special storage management are all significant "complications" beyond simply fixing `longjmp()`.

Another factor bearing on this issue is that, as Martin pointed out in his letter x3j16-ext-29, there is an offsetting simplification that results from deferring stack unwinding until after the handler has completed: there is no need to do a copy of the operand of the `throw`, eliminating the complex storage management issues involved. Since the stack frame of the throw point is still valid, a reference in the `catch` clause can refer directly to the thrown object.

The final issue raised by K&S is contained in section 9:

> Can exceptions be used to handle things like signals? Almost certainly not in most C environments. The trouble is that C uses functions like malloc that are not re-entrant. If an interrupt occurs in the middle of malloc and causes an exception, there is no way to prevent the exception handler from executing malloc again.
>
> A C++ implementation where calling sequences and the entire run-time library are designed around the requirement for reentrancy would make it possible for signals to throw exceptions. Until such implementations are commonplace, if ever, we must recommend that exceptions and signals be kept strictly separate from a language point of view.

There are two problems with this argument. The first is that it assumes something that X3J16 has not yet decided: that C++ will impose no tighter environmental restrictions than are embodied in current C implementations. I don't think that's a reasonable assumption. If the committee adopts a position soon (this year or early next) that libraries and calling sequences must be reentrant, that will give implementors at least two years' notice of the requirement in advance of the effective date of the standard, a period that should be long enough not to inflict undue hardship even on existing implementations desiring to be standard-compliant. Don't forget that we're not just descriptive of current practice -- the very fact that we are considering new features like exception handling is implicit recognition of our prescriptive role. Will "such implementations [ever be] commonplace?" They will if we require them to be.

The other problem is that the argument assumes that the (admittedly draconian) requirements placed on signal handlers are too strict for C++ exception handlers. I claim that, even if the committee chooses to live within the limits of the C environment, the restrictions on signal handlers are no more burdensome in C++ than they are in C -- that is, that someone writing, say, a SIGINT handler would write exactly the same code as if there were an `interactive_interrupt` (resumable) exception available, except that he/she would have to learn the C library interface instead of being able to use the built-in features of the C++ language.

This latter point really gets to the heart of the reason I favor allowing resumption in the exception handling scheme. Adding exception handling to the language is a marvelous opportunity to bring simplicity to chaos, to unify overlapping, redundant, and inconsistent features. I fully expect that exception handling will eliminate the need for C++ programmers to learn about `setjmp/longjmp`, `errno`, and various out-of-band function return values. Given the *fact* that there are applications for resumable

exceptions (I consider that to be beyond dispute; the argument is over whether to support them or not), it seems *much* preferable to incorporate them for a small expense into the standard mechanism than to relegate them to an ad-hoc, mostly redundant parallel mechanism (auxiliary functions like new_handler). And, assuming the existence of resumable exceptions, the redundancy of C signals is glaringly apparent, so that's another C-ish feature like longjmp() that can be subsumed and ignored. Exceptions, resumptions, and signals are all very closely related -- why live with the complexity of three separate mechanisms that do very similar things when one mechanism can do it all?

Having dealt, I think, with all the objections that have been levelled against resumption, at least all the ones I've seen in print, I'd like to conclude this document with a specific proposal. The change to the K&S scheme is small, requiring no syntax changes: all that is needed is to defer unwinding the stack until after the selected handler finishes its execution. Given that one modification, resumption can then be implemented by means of a standard library class, as follows:

```
class resumable_exception {
public:
    resumable_exception();
    int toss();
    int resumption_permitted();
    void resume(int = 0);
private:
    int toss_on_stack;
    struct resumption {
        int val;
        };
    };
```

With this definition (the implementation of which is described below), the floppy drive access example mentioned at the beginning of the letter can be programmed as follows:

```
class floppy_door_open: public resumable_exception { };

void main_loop() {
    for (;;) {
        try {
            // dispatch on events/commands/whatever
            }
        catch(floppy_door_open& excp) {
            if (excp.resumption_permitted()) {
                printf("Floppy door open: retry (Y/N)? ");
                if (getchar() == 'Y')
                    excp.resume();  // will not return
                }
            printf("Floppy access aborted.\n");
            // Falling off bottom of handler => termination
            }   // catch(floppy_door_open&)
        }   // for (;;)
    }   // main_loop()

// ...
```

```
void floppy_write(const char* buff, size_t count) {
    floppy_door_open excp;
    while (floppy_door_is_open)
        excp.toss();
    // Floppy door now known to be closed
    // ...
    }
```

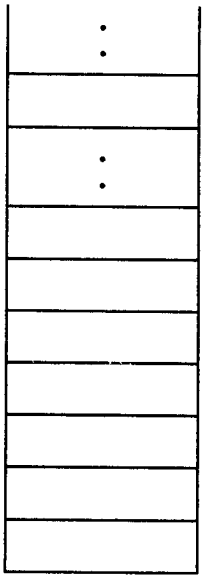
Here is the implementation of the resumable_exception class:

```
resumable_exception::resumable_exception(): toss_on_stack(0) { }

int resumable_exception::toss() {
    toss_on_stack = 1;      // allow resumption
    try {
        throw *this;
        }
    catch (resumption& r) {
        toss_on_stack = 0;  // disallow resumption
        return r.val;
        }
    }

int resumable_exception::resumption_permitted() {
    return toss_on_stack;
    }

void resumable_exception::resume(int val) {
    if (!toss_on_stack)
        terminate();        // illegal resumption
    resumption r;
    r.val = val;
    throw r;
    }
```

A few comments on the resumable_exception class are in order. The purpose of the toss_on_stack data member is to provide a per-throw choice of whether resumption is permitted; simply throwing a resumable_exception does *not* allow resumption. The only way resumption is permitted is by a) using a resumable_exception type *and* b) using the member function toss().

Resumption is accomplished in the resume() member function by throwing the nested private struct resumption. This exception is caught in the toss() member function, which is still on the stack. When the catch clause in toss() exits, the stack is unwound back to the return to the caller of toss(), effectuating resumption.

This process might be clarified with a depiction of the stack after calling resume(). Given the floppy drive example, the stack will look like this (earlier in the stack is toward the top of the picture) immediately prior to the return r.val statement in toss():



The return from the catch clause in toss() will exit the handler, unwinding the stack back into the main frame for toss(), which then returns normally to the call from floppy_write(). Note that resume() must be implemented via throw in order to achieve full generality -- the catch clause in main_loop() might have instantiated destructible auto objects during its processing, and these must be cleaned up prior to the return to the thrower.

The resumption object thrown by resume() has an integer member, used to provide a primitive narrow-bandwidth interface between the catch clause and the throw point. This will be sufficient for many applications (e.g., to distinguish between resumption to retry the action and resumption to ignore the problem, where such a distinction might make sense). Of course, it is possible to put additional data members and member functions into the class derived from resumable_exception in order to provide as extensive an information flow as needed by the application at hand -- since the thrown object can be passed by reference to the catch clause, changes made to it can be interrogated by the throwing function after resumption.

In conclusion, then, my position is that resumption in exception handling is not very hard to implement and meets real needs; I hope we can agree on a specification that accommodates those needs rather than an artificially limited one that discriminates against them. I'll look forward to additional discussion in Seattle.