File : c:\wzmail\files\x3j16-ex.fld
Messages : .

8
From martino Fri Jun 29 10:47:16 1990
To: uunet!bartok.att.com!redbone!x3j16-ext
Subject: Exception Handling
Date: Fri Jun 29 10:47:12 1990

The current proposal looks great. I am very optimistic about it being resolved soon. However I have still two comments to make, both of which I consider serious.

Termination Vs. Resumption

Okay, I have a bee in my bonnet about this one. I can see little justification for outright forbidding it. Granted, there are a thousand ways it could be abused, but exception handling per se, is susceptible to abuse anyway, as are many other constructs in C++ and C.

Here at Microsoft, we have a structured exception mechanism in our internal compilers, which is used for much of the newer operating system work. The new OS work is heavily OO for supporting high level GUI's and abstract file managers. This exception handling mechanism does support resumption, and is essential for efficient recovery in the kernel. I do not doubt that a more aesthetic solution could be found, but so far, attempts to resolve the problems using "aesthetic" solutions, have proven to be inefficient and worse, inflexible in the light of our Dynamically Loaded Libraries. Our structured exception handling is not unlike the proposal, differing in that it supports resumption, all exceptions are integers and there is only one 'catch' per 'try' block. Syntax is different, but not dramatically. It too supports reraise and terminate.

A resumption model also allows for more efficient 'pass-by-reference' exceptions, which can be allocated at the point where the exception is raised, simply preserving the polymorphic nature of abstract exceptions. Although the proposal does not have a built-in special exception base class, I suspect that most developers will use their own special base class, specific to a project. This leads to a model where 'pass-by-reference' is more interesting than the 'pass-by-value' model. With the termination model, this pass by reference mechanism places unusual requirements on a heap manager, especially in the event of 'out-of-memory'.

With each proposal, Bjarne has enlarged the section describing why resumption is awful. I am not convinced. However, I do agree that in some cases (perhaps many cases), resumption is not only meaningless, but probably dangerous too.

I have a proposal for this. I suggest we introduce two kinds of exception raise (throw). One raises an exception which is NOT permitted to resume, and the other raises an exception which IS permitted to resume. The former can be processed in accordance with the current proposal, the raise point knowing that it cannot return; with the exception that stack unwinding be done at the end of the

handler, and not on entry to the handler. The second case is also easily handled by a compiler, as it knows that the exception may resume, and can acknowledge the possibility of execution continuing.

Any attempt to resume from a non-resumable error, would result in a run-time error, or perhaps the calling of a reserved handler, like terminate or abort.

Cleanup of Partially Constructed Objects

A derived type, or a type which has an embedded member of some other type, is not not a case of "The sum of the parts". It is a completely new type. A derived type "is a kind of" a base type. It is not semantically a concatenation of base types. I consider that a type which has been partially initialised, is simple NOT initialised, and its parts cannot be considered to be objects in themselves.

This gets worse when it interacts with friends or derived classes. For instance, a class may provide provide public construtors, which always leave the object in a valid state for destruction on correct completion. However, the same class may also provide protected constructors, which do not completely initialise the object, but are intended to be completed by a deriving class. In fact, I had to use just such an approach when working on CommonView at Glockenspiel. One of the types was a type called 'ScrollBar'. This was a 1.2 C++ variant of an abstract class, never intended for standalone existance. It only had a protected constructor. From the 'ScrollBar' type were derived a 'HorizontalScrollBar' and a 'VerticalScrollBar'. Since the base abstract class did not have enough information to commit the form of the actual scrollbar, this had to be left to the derived class, and the registration of the object to the underlying window manager had to be deferred. However, once the derived object had completed its construction, the abstract base object had all the information necessary to correctly remove the scrollbar from the windowing environment. If the derived object failed to complete its construction, the abstract base object would not be in a valid state for destruction, and could cause the windowing system to crash if attempted. Another case involved the way in which a Dialog window differed from a regular window. The problem here, was the registration of the event (message) handler. The handler for a regular window was totally different to the handler for a dialog window in the underlying windowing environment. This required the deferring of this decision to later in the derived object's context, again resulting in enough information known in the abstract class for correct destruction.

In both these cases, the derived object could only be considered complete if the complete construction had finished, and thus the concept of a 'partial' object does not exist.

This kind of interaction can occur quite easily with both protected access, and friends, where certain normally invalid states could be achieved by an object in the context of it being a base or member of another object.

Irrsepective of the religion of termination/resumption, I consider the second case to be simply an error, and must at all costs be removed from the

proposal. Thus a object (and all of its parts) is only eligible for destruction during unwind, iff, its construction has fully completed.

Incidently, the same is true of global objects allocated in static store. The are considered for destruction in the whole, and not in the part. There is no semantic difference between the integrity of automatic objects and static objects.

A similar claim can be made for partial destruction.

The rest looks great, I have no other gripes about the proposal, and envision no great problems in implementing it (I won't be using 'setjmp/longjmp' however).

See you all in Seattle on the 9th. Sorry to see that Rob Seliger can't come.

Martin