WG21/N2273 **J16/07-0133** **2007-05-01**

Nick Maclaren
University of Cambridge Computing Service,
New Museums Site, Pembroke Street,
Cambridge CB2 3QH, England.
Email: nmm1@cam.ac.uk
Tel.: +44 1223 334761
Fax: +44 1223 334679

Non-Memory Actions (Core Aspects)

1.0. Introduction

This is another go to try to explain some aspects of the non-memory actions, and how they cause trouble with concurrency.

It relates to standard- or implementation-defined behaviour and side-effects that are not clearly modifications to or accesses of a memory location, hidden or otherwise. It includes objects that have strange properties, of the sort that are not normal for a simple object stored in memory.

Note that this is **not** a purely theoretical problem, as I have seen it cause trouble on a fair number of systems. A great many implementations (at all levels, from hardware up to libraries) remember to honour the synchronisation and serialisation properties for memory accesses, but tend to forget about the non-memory ones. This is not helped by most languages that mention them effectively specifying them as undefined behaviour, because that makes it almopst impossible for customers to report problems as bugs through the usual support interfaces.

1.1. Examples and the Problem

C++ as of today has relatively few that have any effects outside the library, though several in that, but POSIX and C99 introduce many more. Here is a partial list of the most relevant ones relevant to the core language, of which only the first is actually required in C++ implementations today, but where the second is **required** for C99 and POSIX 2001 support (see 1.2).

1: C++ exceptions. These are objects with some extra restrictions, a certain amount of hidden state, and the property that rethrowing an exception could be regarded as a modification action on that hidden state.

2: IEEE 754 exception flags and modes, or at least the subset of features supported in C99.

3: Code-generated interrupts (e.g. `SIGFPE`), whether handled as C++ exceptions, signals or otherwise. The Language Independent Arithmetic standard should also be mentioned.

4: Some of POSIX concurrency modes, most especially PTHREAD_CANCEL_ASYNCHRONOUS, signal handling and the thread scheduling options.

5: WG14 Technical Report 18037 provides some hardware I/O register support for C99, but I have no idea what C++'s position on it is.

6: Implementation-specific versions of the above, which are probably more widely used than the standardised ones, and other implementation-modes, states, counts and flags. A class that is extremely important to many users and often gets forgotten is hardware counters (including IEEE 754 exception counts).

Some of those are apparently specific to a particular thread, but that is deceptive. POSIX allows the handlers of code-generated interrupts to be handled in a newly created thread, and there are other, more esoteric, issues that can affect some people. E.g., for arcane hardware and operating system reasons, some issues can impact a whole system (let alone a process!); this is not obvious even to the library implementor, let alone the developer, but is clearly of great importance to real-time and embedded developers, and even HPC ones.

There is also the problem that the handler of an event (whether a C++ exception or some other kind of event) may do something that is visible to other threads or a non-memory action may become visible in other ways. For these reasons, such actions need some suitable wording. The remaining sections refer to wording that I think will be needed.

1.2. C++ Exceptions

As far as I know, the hidden state is entirely concerned with which `catch`/`try` block are executed and in what order, so is not directly visible to the program. Therefore the only question is whether the exception object should be accessible to other threads, and what should be said about that. Whether or not it is, some changes to 15.1 will be needed.

My personal view is that it should not be, largely because that makes life easier for the implementor, and my guess is that the wording changes will be easier. I can't see any major problems either way, but leaving it unspecified will cause trouble, because some developers will assume that it is accessible from other threads, and some implementors will assume that it need not be.

Proposal: *It should be undefined behaviour to access a C++ exception from any thread except the one in which it was thrown.*

1.3. IEEE 754 Flags and Modes

This is too complicated an issue for me to cover here, but I will try to write another paper on the details.

The current draft of C++0X is seriously confusing, in that it includes the function interfaces of C99 but not the language changes. And, despite being in the library, C99 <`fenv.h`> is really a compiler feature, with its primary consequences being on code generation.

I believe that this needs to be clarified, though I am not at all sure what should (or even can!) be said. What I can guarantee is that virtually every implementor will choose a different interpretation of the C++0X standard unless **something** is said.

Proposal: *This should be regarded as an outstanding work item, but only to clarify what the C++ standard defines.*

1.4. Other Non-Memory Actions

I do not believe that there are any other non-memory actions that are suffiently well standardised to even attempt to draft normative text, but I can witness that the lack of any requirement to document the details of how other standards interact already causes serious problems, and hence worse ones to developers.

Proposal: I believe that some informative wording should be included, such as the following:

> *Where an implementation includes extensions that have side-effects (in the most general sense), as far as is practical an implementation should define the synchronisation of such behaviour in terms of the basic memory model.*