

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

The Memory Model and the C++ Library, Non-Memory Actions etc.

1.0. Introduction

This is an attempt to describe how various actions can be bound to the C++ memory model. The wording in this document is a complete crock, because the author is neither a C++ programmer nor familiar with C++ standard notation, so please do not regard it as a good draft of an actual specification.

It includes wording that is intended to cover more than is in the current C++ standard. This is because it is essential to give guidance to those extending C++ if those extensions are not to conflict with each other, and even the intent of the C++ standard. Existing, important uses and proposed extensions include LIA-1, IEEE 754, POSIX, MPI, OpenMP and many others that the author knows about but is not familiar with, such as database interfaces.

Most of these are in the form of external libraries, and add few problems that are not already in C++, except as for the following aspects:

- They very often use C++ features that are not currently used in the C++ Standard Library interfaces.
- Asynchronicity, including asynchronous I/O (POSIX and other), message passing (including MPI non-blocking calls), and so on, almost always of the sort where the asynchronous action can be bound either to a flow of control (a thread) or to a specific object.

[*Experience is that such controlled asynchronicity is manageable; the general case is much harder and is ignored in this document. The author has implementation experience of it in the context of a von Neumann language, and baulks at adding it to a CPL-derived language standard.*

One must also not forget the buffer passed to setbuf and setvbuf, which has memory consistency semantics as if it were being used asynchronously.]

- Exceptions in the normal computer science sense, including LIA-1 and IEEE 754R floating-point flags and traps, POSIX signals, their equivalents on other systems (e.g. zOS abends), and so on.

Obviously, this document does not propose that those extensions should be added to C++, but it seems reasonable to have a framework of how they should map to the C++ memory model, for guidance to implementors that wish to do so. Also, in many of those cases, there are proposals for such extensions and groups working on them, and POSIX is used from a great many important C++ programs.

[*Let's not get started on GUI interfaces, which are often used from C++, and would be ideal candidates for threading if they were not so unspeakable. Some kind of a C++ framework could only help to improve them, but adapting C++ enough to permit them would destroy much of C++. For example, many objects in important GUI interfaces may be accessed both from synchronous (main thread) code and from asynchronous event handlers, with no requirement for any explicit synchronisation actions. At this point, one may laugh, cry or scream, to taste. Here, I merely close my eyes to that issue and hope that it will go away.]*

Unless stated otherwise, all references are to ISO/IEC 14882:1998 (C++), IEEE Std 1003.1 2004 (POSIX) or ISO/IEC 9899:1999 (C99). LIA-1, (ISO/IEC 10967), IEEE 754 (ISO/IEC 10559), MPI (see <http://www.mpi.org>) and OpenMP (see <http://www.openmp.org>) are also referred to.

1.1. Basics of the Approach

Library function calls, dynamic library actions, other state access and changes, non-memory actions and even exceptions are mapped to (possibly composite) virtual memory actions, for the purpose of defining threading consistency. Unless stated otherwise, all such actions have sequenced-before constraints and relations exactly as if they were memory actions of a similar type (i.e. an ordinary read or update).

A non-memory action will behave like a read only if it is permitted to occur multiple times between a pair of sequence points, and this is more restrictive than being a “read-like” action. This has to be flannelled a bit, because the sequence point specification is such a mess, but indicates the intent.

[*This does not seem to be required by current C++, but the condition is critical if there is ever to be a C++ binding to certain other standards. For example, catching an exception (such as a POSIX signal) is **not** a read-like action.*]

This document distinguishes between the C++ standard library functions that are not in the C Library and those that are. To give some flexibility to implementors, it also distinguishes between those included via the headers in C++ sections C.2 and D.5. See under the section on the C Library for why; it is not a pretty problem.

For sanity, it assumes that the memory model will define the following concepts, and that any specification of an API will include at least the latter two (though that may be naïve):

1. A full simplex synchronisation action between threads A and B, such that everything in thread A that is sequenced before the action happens before everything in thread B that is sequenced after the action.
2. A full duplex synchronisation action between threads A and B, such that everything in either thread that is sequenced before the action happens before everything in either thread that is sequenced after the action.
3. A full barrier synchronisation action between all threads, such that everything in any thread that is sequenced before the action happens before everything in any thread that is sequenced after the action.

I don't think that it is necessary to forbid actions that are sequenced after the synchronisation actions from being brought forward, except as forbidden by the main memory model, but that needs more checking.

1.2. Aggregate Objects

The first one is a slight conflict between the body of the language and the library, and shows up in a similar conflict between Hans Boehm's document and this one. It is whether aggregate objects (usually arrays) passed as arguments to library functions should be treated as unitary or as a collection of subobjects. There are three main arguments:

- The traditional C (and C++) model is that the basic operations are on basic types, and their object model uses that assumption.
- This was changed, both explicitly and implicitly, by ISO C and C++, for good software-engineering and performance reasons.
- It seems a bad idea to have wildly separate semantics for containers, opaque aggregates and aggregates with a visible type (see below).

Let's consider just the issue for floating-point arrays (created by `valarray`). Most current hardware does the simple cases and expects the harder ones to be fixed up by emulation. One of the traditional and efficient implementations of array operations is to check once at the end, and emulate the action on the whole array if needed. Now, this means that data may be stored twice. There are similar possibilities in `<basic_string.h>`.

In Hans Boehm's model, any program that can detect that is undefined, unless the basic type is atomic (i.e. `_async volatile`); not a major problem. But it seems to be a bad idea to add an unnecessary trap for programmers and implementors, so this document takes the approach that actions on aggregate objects act on the whole object, as far as memory synchronisation rules are concerned. Fortran uses that model, very successfully.

The alternative approach would be possible, but it would mean a lot more wording for some library calls, and introduces data-dependent memory synchronisation conditions within library calls — e.g. **exactly** how many bytes are `strcat`, `fprintf`, `fscanf` and (worse) `strtok` specified to access, and how, as far as the memory model is concerned?

1.3. Containers and Similar

Consider structures like the one created by the following:

```
typedef struct { double *ptr; int len; } fred;
fred joe[100];

for (i = 0; i < 100; ++i) {
    joe[i].len = rand()+1;
    joe[i].ptr = (double *)malloc(joe[i].len*sizeof(double));
}
```

Hans Boehm's memory model makes it clear that operations on `joe` may be performed in parallel, provide that they don't clash on the actual elements referred to. So far, so good. But should that still apply when `joe` is a container with an opaque implementation (such as those in the C++ Standard Library)? And should the semantics be permitted to be different if, say, the `vector` class is implemented as an array or as a height-balanced tree? If so, we have one of two choices:

- Whether programs have defined behaviour or not depends on unspecified properties of the implementation.
- The specification must state exactly how the implementation behaves.

Neither solution is good — indeed, in my view, neither is acceptable. This leaves us with two choices, where the decision could be made separately for each container class, or even each method within a class:

- The actions are on the container as a whole, and unserialised incompatible actions (e.g. two updates) on the same container object in two threads is undefined behaviour.
- The class is required to implement the actions as if they were atomic (i.e. to use locking where needed).

The first is likely to be considerably faster, but the second is considerably more useful. In fact, it would be perfectly reasonable to either replicate the classes or use the template mechanism to provide both sets of semantics.

My understanding of C++ is that it would be natural to choose the first (unconditionally) for the general utilities, strings and numerics libraries (see below for streams and stream buffers), and the real issue is what to do about the containers library. The iterators library would use the semantics of the construct it was iterating over, I assume, but that needs checking.

1.4. Templates

This includes anything with class arguments, if there are any other such constructions in C++. My understanding of such things is inadequate to suggest any approach — there may be nothing to specify, but I mention them merely because I cannot convince myself that such things as the initialisation of static members have no threading implications beyond the obvious.

1.5. Streams

I defer a discussion of these until later.

2.0. Specific Proposals

While the following may be controversial, I think that there is less choice than might appear, and the following proposals follow almost inevitably from the existing C++ model, Hans Boehm's memory model, and the assumed requirement to make some kind of sense in combination with POSIX-style threading.

POSIX is **not** used as a model, for reasons described in my other document and summarised later, but this should be compatible with the defined aspects of POSIX threads.

2.1. Thread Safety in the C++ Library

Absolutely thread safe is when a function is defined to operate using solely thread-local data and state, objects accessed by its arguments or specified named objects. Whether two such functions can be called concurrently in different threads is determined solely by the memory consistency rules on those objects.

Conditionally thread safe is when a function is defined to behave as it it were absolutely thread safe in the absence of exceptional conditions. However, calling two such functions in two threads may cause either or both to fail, whether by returning an error indication, throwing exceptions or otherwise.

Not thread safe is when a function is not defined to be either absolutely or conditionally thread safe.

Two hidden objects are **connected** if they are the same hidden object, or sub-objects of a single hidden object, which may be internal or external (see under External Actions). Two functions are connected if they access the same object or two connected objects. Connected functions are never absolutely thread safe.

[*Global data that is set up before any program code is entered and is never changed is ignored.*]

For example, any function that may use memory allocation internally, or may use any variable global state, can be at most conditionally thread safe.

Some functions in the C Library return objects that other standards specify may be either reused or may be new in each call (e.g. `getenv`); these are classed as not thread safe, following POSIX.

Unless otherwise specified, the following will hold:

1. A program that accesses two connected objects, of which either access is an update, in two threads without separating them by a full duplex synchronisation action is ill-formed.
2. A program that calls a not thread safe function, or two connected functions either of which is not thread safe, in two threads without separating them by a full duplex synchronisation action is ill-formed.

[*The last two conditions may seem draconian, but could make a massive difference to implementability.*]

The main connexions in the C++ library are:

3. The stored global locale is a single atomic hidden object, which is read in its entirety, precisely once per call, by all functions that use any aspect of that stored locale. `global` from `<locale>` and `setlocale` from `<locale.h>` and `<locale.h>` update that object, writing to it precisely once per call.

[*It would be better to make all functions that use the stored locale not thread safe, but that would be a major clash with POSIX. The loss of efficiency for locale-using functions is tolerable, unlike that for `cin`, `cout` etc. Note that the above means that updating it in parallel is not defined but using it in parallel with an update will get either the old or the new value.*]

4. The `<iostream>` objects `cin` and `wcin` and the `<stdio.h>/<stdio.h>` object `stdin` are connected, as are `cout`, `wcout` and `stdout`, and `cerr`, `clog`, `wcerr`, `wclog` and `stderr`.
6. All file-based streams, `FILE` objects and functions that take a file name as an argument that refer to the same file are connected by their external object (see under External Actions).

There is one nasty but important omission from the above: asynchronous signals. That is discussed in the relevant section.

2.2. Default C++ Standard Library Call Semantics

Unless otherwise specified, the specification of a library function call or similar action is mapped to virtual memory actions as if the function behaved in the following way, irrespective of its actual behaviour in any particular call:

1. All `const` argument and named objects are read in their entirety. In the case of an argument that is a pointer but the object pointed to is used, the latter is read.

2. All other argument and named objects are updated in their entirety. In the case of an argument that is a pointer but the object pointed to is used, the latter is updated.
3. All accesses to argument and named non-atomic objects occur simultaneously.
4. All accesses to argument or named atomic objects occur an indefinite number of times in an indefinite order. If the permissible variations affect the results in any way, except as explicitly permitted by the specification, the program is ill-formed.

[*This is currently unnecessary, but it is trying to define a model which allows for future expansion, including the inclusion of an atomic library! Note that this requires a very precise specification of any library function that uses atomic objects.*]

5. No other form of sequencing or thread locking is performed within the function.
6. Where a call is specified to set a state that changes the behaviour of another call or language feature, or save an rvalue for later use, it updates a hidden object.
7. Where a call is specified to save an lvalue for later use, it updates both a hidden object and the object defined by that lvalue.

[*This is needed because the C++ standard allows the the implementation to copy read-only objects.*]

8. Where the order of calling two such functions is specified to be significant, explicitly or by implication, on subsequent function calls, they update (possibly sub-objects of) the same hidden object.
9. Where a call uses such a saved state or rvalue, it accesses that hidden object.
10. Where a call uses such a saved lvalue, it accesses both that hidden object and the object defined by that lvalue.
11. Where such calls may be used an arbitrary number of times between sequence points, the access is a read action, otherwise it is an update.

[*Note that, as far as the above condition is concerned, $f()+g()$ invokes two calls between a pair of sequence points. The intent of the C and C++ standard is almost certainly that they are sequenced in an unspecified order (though its wording is ambiguous), but this condition assumes that they may be interleaved for the purposes of exposition.*]

2.3. Allocation, Deallocation etc.

1. Implicit allocation and deallocation of memory behaves as if it were performed by calls to the appropriate library functions with the calls from the appropriate thread, and sequenced like a normal operation.

[*Yes, I know that the C++ standard already says this!*]

2. All memory allocation and deallocation (destruction) actions are conditionally thread-safe in the sense defined below, except that one thread's allocation of memory may cause it to be inaccessible to other threads until it is both deallocated and that fact is communicated to other threads.
3. At a full global barrier action, all deallocated memory becomes available for use by any thread. When thread A performs a full simplex or duplex synchronisation action with thread B, then all memory that has been deallocated by thread A since becomes available for use by thread B. Obviously, 'deallocated' in this context refers only to memory that has not been reallocated since deallocation.

[*This one is often forgotten. Many standards and implementations have temporary dynamic leaks — which is unavoidable — but do not specify any method of clearing them! It is a **very** heavy implementation penalty to insist that it becomes available immediately upon deallocation. The above is an arbitrary set of rules, and many others would work.*]

2.4. Exceptions and Events

The following conditions refer to C++ exceptions, as defined in chapter 15 of the C++ standard, whether they are raised by a throw statement, by compiled code or the standard library.

1. All exceptions shall be thrown in the thread where the action causing the exception occurred, and sequenced as part of that action.

[*The IBM System/390, the DEC Alpha, the Intel/HP Itanium and many other architectures may raise exceptions asynchronously, unless the implementation takes special action to stop that.*]

2. There is a sequence point between all actions (including all memory accesses) between the point the exception is thrown and the calling of the handler. Also, from the viewpoint of thread safety, a function call terminates when it returns or throws an exception.

[*Without this, there is a ghastly problem deciding when accesses will complete, if ever. Because exception handling (even the C++ form) is potentially out-of-band, the sequence point rules in C++ 1.9 paragraph 17 are not enough. Actually, they are not enough even at present, because throwing an exception in a function with a handler outside it means that no returned value is copied, so footnote 11 does not apply.*

See below for a discussion on asynchronous signals.]

3. A program that accesses the exception object from any thread except the one that created it is ill-formed.

[*While this may be unnecessary, one really does **not** want to worry about threading, exception handling and destructors simultaneously, let alone garbage collection.*]

The following refers to more general exception and event handling, including such mechanisms as IEEE 754 or LIA-1 traps, POSIX signals etc., and is necessarily in the form of recommendations:

4. Where an exception or event is raised because of an action in a specific thread, an implementation should attempt to handle it in the same way as a C++ exception, where possible.
5. An implementation may define circumstances under which an exception or event is thrown in one thread because of an action in another, raised against the program as a whole, or raised asynchronously, and shall then define the semantics of such exceptions and events and their handling.

[*This is necessary because one of POSIX's modes allows the nomination of a thread to receive signals, and forbidding such behaviour would prevent a C++ implementation from supporting both threads and exceptions on such a system. There are also circumstances where an exception may occur with no identifiable thread causing it, and it is critical if anyone were ever to develop a high-performance, high-RAS streams or I/O capability.*]

2.5. Asynchronous Signals and Events

Note that these include signals sent from other processes, kernel generated signals like SIGCHLD, events raised by several GUI interfaces, MPI one-sided communication, and the message passing mechanisms used in several threading models.

Since time immemorial, the safest and most portable way of receiving asynchronous signals and events in applications has been to trap them, set a flag, and continue. The main thread then checks the flag at places it is safe to abandon what it is doing, and does what is appropriate. A huge number of very important daemons and other applications rely on this. Let us call this interrupt handling.

The alternative was to mask off all signals in critical code, which was (and is) the favoured approach in device drivers and similar kernel threads, but it introduces major problems when used in applications. For various reasons, POSIX chose to canonise this model and deprecate the other — it has been fairly successful in privileged daemon code, but not as much as its proponents claim. However, POSIX still supports the interrupt model and, as said, it is heavily used.

Many languages and systems have had a composite exception, signal and event model, with the difference between interrupt handling and C++-style exception handling being whether the handler cancels the exception and returns transparently, or whether it aborts the execution stack up to a certain level. While I am not proposing that C++ should support such use, there is a question of whether the C++ Standard should **forbid** implementations from extending exception handling in that way.

There is essentially only one issue: whether functions are required to clean up before calling a handler or are permitted to do so afterwards. The former is needed if the handler is to do anything significant, and the latter is needed if it is to return transparently, as implementors who have done it can witness. My view is that the current zeitgeist is such that it would be a bad idea to even **permit** C++ extensions to mix

interrupt and C++-style exception handling, which is why it was locked out above. But there is a case for adding the following:

1. An implementation may define a mechanism by which asynchronous signals and events are trapped, the execution of a thread is interrupted, the handler sets an atomic flag or flags, and execution continues at the point of interruption with the program state unchanged except for the setting of the atomic flag or flags. In such a case, the memory model will ignore the call to the handler, and treat the update of the atomic flags as if it had been done by some other, unsynchronised, thread.

2.6. External Actions, plus Asynchronicity

Interactions with the environment (including all actions on objects created by constructors or functions in C++ section 27.8, the I/O functions in `<wchar.h>`, and `getenv`, `system` etc.) should be implemented as if they consist of a normal C++ class that handles the internal aspects, which is a derived from an opaque class that handles the external aspects.

An implementation shall support the normal C++ semantics for the first component, entirely within the invoking thread (without prejudice to the decision on what a thread is), and should do so as far as is reasonable for the second one. The following are some specific recommendations on how to handle particular issues.

1. Where it makes sense, the external action should be handled synchronously and entirely within the invoking thread.

Where it does not, an implementation should attempt to use the following approach:

2. If the action is asynchronous or can have multiple, simultaneous instances, a handle should be defined for the action, preferably an object. A concrete example might be a FILE object.
3. Actions and events (including exceptions) should be serialised on that handle. For example, an asynchronous exception should be raised when the next action on that handle takes place, in the thread that invoked that action, as if it were detected by polling and not interrupt.
4. A well-formed program shall call all functions using such an asynchronous handle as if they were not thread safe.

[*This is to allow an implementation a decent degree of flexibility, for performance.*]

5. An implementation should provide some action on a handle that synchronises all its outstanding asynchronous actions and raises any resulting exceptions and events.
6. Where an action is asynchronous and applies to an object (i.e. one with an lvalue), the consistency constraints on a program are as if that object were being read or updated in its entirety continually during that period. Note that the object and the handle may be different (e.g. an I/O buffer, and its FILE object or a request handle).

[*Experience is that the above level of controlled asynchronicity is both manageable and implementable. True, general asynchronicity is beyond the mindset of most programmers and even most implementors, despite the fact that GUIs use it.*]

External operations used for communication, such as I/O, signalling and message passing, should be implemented in such a way as to provide synchronize-with constraints that are compatible with causality. In particular:

7. Where an external action reads the state of an external entity, it should be associated with memory acquire semantics.
8. Where an external action changes the state of an external entity, it should be associated with memory release semantics.
9. Where thread A changes the state of an external entity to a value different from the previous one, thread B reads the state of that external entity and receives the new value, then thread A *synchronizes-with* thread B.

[*A stronger condition would be nice, but it would also cause implementations real headaches.*]

10. The above rule should be applied to cases where thread A changes the state of external entity X, one or more external agents use that change to change the state of external entity Y and thread B reads the changed state of Y.

2.7. Streams

This is a problem area. As those with mainframe experience can confirm, the key to decent streaming I/O performance is to permit asynchronicity. This is done largely by removing any semantic meaning from transfer boundaries; it is possible using the transfer operations in C `<stdio.h>`, but not with POSIX `read` and `write`. I have not convinced myself what the semantic model of the C++ streams library is yet.

In particular, one traditional, natural and efficient way of implementing streaming output on a threaded system is for the main thread to feed data into a FIFO and for an auxiliary thread to remove data from that and write it to wherever it is going. And similarly for input, with the auxiliary thread reading ahead far enough to keep the FIFO full.

There is a problem with exceptions that occur in the auxiliary thread, especially when the speculatively read data is not in fact read by the main thread, but that is soluble by binding such exceptions to the stream, as described under External Actions above. Given that, it is possible to achieve near-optimal throughput, using only moderate buffers, and with excellent reliability of exceptions.

However, it is critical not to introduce synchronisation by accident, as a single synchronised stage in a long asynchronous pipeline can destroy the efficiency of the whole pipeline.

My question is whether C++ regards allowing threaded implementations to do that with C++ streams is a desirable objective. Whether or not that is so, then there will probably be some consequential changes to the wording, to clarify the intent. But there may need to be some changes of specification if full, asynchronous streaming is to be supported.

What is the C++ viewpoint? Is streaming performance a major objective?

2.8. File-based Streams and Diagnostics

There are a few things that are not obvious (except to an implementor), but are essential:

1. For the purposes of the memory model, FILE objects are not atomic, and are updated by every access whether or not it can change their state.
2. For the purposes of the memory model, the external object associated with the actual file is not atomic, and is updated by every access whether or not it can change its state.

The above conditions say that two threads writing to `cerr` in parallel is undefined behaviour (unlike what POSIX says for C Library calls). However, there is a case for providing a simple, inefficient, serialised way of producing diagnostic output. Experience indicates that such a mechanism is almost essential for debugging, and that many users have difficulty in writing such a thing.

The minimum requirement is that each thread can write a short, single line, message in such a way that the actual lines remain intact. This is easy to specify as a single function call, but it is messy if one starts from the C++ file-based stream model. However, it could be done.

3.0. The C Library

This describes what should be done about the C Library as included in C++ — which I know rather better than the rest of C++! It specifically does **not** refer to Standard C, as that has a huge number of extra problems (at least when we get to C99).

One of its targets is to tolerate implementations of C++ that use the native POSIX/C library, either for all functions called via headers in C++ sections C.2 and D.5 or for only those called via ones in D.5. I believe that is essential.

3.1. POSIX 2003

POSIX is, at best, unhelpful. The most relevant definitions seem to be:

Thread-Safe: A function that may be safely invoked concurrently by multiple threads. Each function defined in the System Interfaces volume of IEEE Std 1003.1-2001 is thread-safe unless explicitly stated otherwise. Examples are any “pure” function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads.

Concurrent Execution: Functions that suspend the execution of the calling thread shall not cause the execution of other threads to be indefinitely suspended.

To keep this short, this document will not go into the assumptions that the above is making, why they are unreasonable and even incorrect, and other consequences (including that the question of whether any particular function is thread safe is indeterminable, even in theory). There is another document on those. It will make only three points in summary:

- POSIX-style thread safety is not absolute, and implies only that the concurrent execution of thread safe functions is not undefined behaviour (though it may well be unspecified behaviour, and may cause program failure).
- By requiring almost all functions to be thread-safe, it is turning its back on scalable performance. This is a repeat of mistakes made previously with other facilities, like I/O, asynchronous I/O and threading itself.
- There are internal inconsistencies, ambiguities and insanities, including where several functions are described as thread safe when they are clearly not.

As a result, this document maintains consistency with POSIX only where POSIX looks plausible in a C++ context: “as close to POSIX as possible, but no closer.”

3.2. Choice of Boundaries

POSIX thread-safety is close to conditional thread safety, but there are the issues described above, as well as several potential implementation nightmares. For example, consider a C and a C++ thread, each allocating memory or performing I/O on the same stream with no explicit synchronisation between the two. For this reason, there are only three alternatives:

- To let POSIX (and C99) impose semantics on C++ — this cannot easily be limited to the C Library, but will affect the remainder of the library and even the base language.
- To have the C++ standard specify that all of the functions in the C Library have different and potentially incompatible threading and memory model semantics to POSIX.
- To separate the C++ Standard Library into three sections: the C Library headers described in C++ section D.5, those described in C++ section C.2 and the native C++ library (the rest).

[*Different implementations maintain their C++ and C interoperability in different ways, so that it is essential to distinguish all three sections. Frankly, any program that mixes any two of these without separating them properly is asking for trouble, and will probably get it.*]

This document takes the last approach. In the paragraphs below, the term “uses an entity” refers to calling a function or accessing an object. Objects and functions are connected as described elsewhere. The following constraints apply to the C Library (whether included by the recommended headers or the deprecated ones), and are in addition to any implied by the rest of this document:

1. A program is ill-formed if it uses a C Library entity and a connected C++ Standard Library entity (not in the C Library) in different threads without separating the two by a full duplex synchronisation function.
2. A program is ill-formed if it uses a C Library entity included by a header in C++ section C.2 and a connected C Library entity included by a header in C++ section D.5 in different threads without separating the two by a full duplex synchronisation function.
3. These constraints apply even when the C++ standard refers to C++ and C library functions being equivalent.

3.3. Specific Conditions on the C Library

The following applies to the headers in C++ section D.5 as well as those in C++ section C.2.

1. All macros and functions from headers defined in C++ section D.5 are connected to the ones of the same name from headers defined in C++ section C.2.
2. All macros and functions from `<cassert.h>`, `<cctype.h>`, `<cerrno.h>`, `<cfloating.h>`, `<ciso646.h>`, `<climits.h>`, `<cmath.h>`, `<cstdarg.h>`, `<stddef.h>` and `<cwctype.h>`, all except `wscoll`, `wcsxfrm` and the I/O functions from `<wchar.h>`, all except `strcoll`, `strerror`, `strtok` and `strxfrm` from `<cstring.h>`, `abs`, `bsearch`, `div`, `labs`, `ldiv`, `qsort`, `strtod`, `strtol` and `strtoul` from `<stdlib.h>`, and `clock`, `difftime`, `mktime` and `time` from `<ctime.h>` are absolutely thread safe.

[I/O functions are those that have a `FILE *` argument, return a `FILE *` result, use `stdin`, `stderr` or `stdout` or refer to an external file by name. Note that the logic of the above is to exclude anything that depends on locales or external actions. The former is necessary in case a global locale is used and one thread corrupts it. The above wording absolutely **must not** be extended to C99, as some of the new facilities in `<fenv.h>` have cross-thread implications on `<math.h>` on many systems, and there are probably other 'gotchas'.]

3. The functions `localeconv` from `<locale.h>`, `raise` and `signal` from `<csignal.h>`, `tmpnam` from `<stdio.h>`, `abort`, `atexit`, `exit`, `getenv`, `mblen`, `mbtowc`, `rand`, `srand`, `system`, `wcstombs` and `wctomb` from `<stdlib.h>`, `strerror` and `strtok` from `<cstring.h>` and `asctime`, `ctime`, `gmtime` and `localtime` from `<ctime.h>` are not thread safe.

[This includes fixes to POSIX's list: `abort`, `exit`, `mblen`, `mbtowc`, `srand` and `tmpnam`. We need to include `raise`, `signal` and `system` because, while a POSIX system might just behave sanely with them, it is an intolerable requirement in general. Calling `abort` and `exit` (or either of them and `atexit`) in parallel is obviously insane.]

4. All other functions in headers in C++ sections C.2 and D.5 are conditionally thread safe.
5. There is a separate copy of `errno` in each thread, and it is initialised to zero at thread creation.
[POSIX does not seem to say what its value is initially; it could be either inherited or cleared.]
6. Using `longjmp` with a `jmp_buf` with contents created by a call to `setjmp` in a different thread is undefined behaviour.
7. Using any function in `<cstdarg.h>` on an `va_list` object created in another thread is undefined behaviour, as is using.

[Note that some threading models do have a decent concept of a thread being rooted at a point in another thread's call tree, so this is not automatically excluded by existing wording. But we **really** want to exclude such tricks!]

The following are special rules for defining the objects used for deciding memory consistency, and apply irrespective of the amount accessed in any particular call:

8. Where a function uses a pointer argument with a separately defined maximum length, the object is that number of bytes starting at the argument address.
9. Where a function uses a null-terminated string argument defined as `const char *`, the object is the whole of that string, including the terminating null character.
10. Where a function uses a `void *` or `char *` argument with no separately defined length, the object is the number of bytes actually written starting at the argument address.

[This is a performance nightmare on some systems, as it means that separate threads can write to consecutive bytes of an array, and the implementation must get that right, penalising the sane programs that do not do that.]

11. With functions `strcat` and `strncat` from `<stdio.h>` and `wscat` and `wcsncat` from `<wchar.h>`, the entirety of the first argument is treated as if it were updated.
12. With functions `setbuf` and `setvbuf` from `<stdio.h>`, the buffer is treated as if it were being continually updated from that point until the stream is closed or the buffer is replaced.

13. With functions `fprintf`, `fscanf`, `printf`, `scanf`, `sprintf`, `sscanf`, `vfprintf`, `vprintf` and `vsprintf` from `<stdio.h>` and functions `fwprintf`, `fwscanf`, `swprintf`, `swscanf`, `vfwprintf`, `vswprintf`, `vwprintf`, `wprintf` and `wscanf` from `<wchar.h>`, only arguments used in the actual call are involved in memory consistency, but are treated as described above when they are.
14. With the function `strtok` from `<stdio.h>`, the first argument is treated as if it were updated in its entirety, both for the initial call and for any subsequent calls using a null first argument.