6 Basics [basic]

6.9 Program execution [basic.exec]

6.9.2 Multi-threaded executions and data races [intro.multithread]

6.9.2.1 General [intro.multithread.general]

[1] A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread.

[*Note 1*: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — *end note*]

Every thread in a program can potentially access every object and function in a program.[38] Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads.

[*Note 2*: Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — *end note*]

Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

[2] For a signal handler that is not executed as a result of a call to the `std::raise` function, it is unspecified which thread of execution contains the signal handler invocation.

38) An object with automatic or thread storage duration ([basic.stc]) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference ([basic.compound]).

6.9.2.2 Data races [intro.races]

[1] The value of an object visible to a thread *T* at a particular point is the initial value of the object, a value assigned to the object by *T*, or a value assigned to the object by another thread, according to the rules below.

[*Note 1*: In some cases, there might instead be undefined behavior. Much of this subclause is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

[2] Two expression evaluations *conflict* if one of them modifies a memory location ([intro.memory]) and the other one reads or modifies the same memory location.

[3] The library defines a number of atomic operations ([atomics]) and operations on mutexes ([thread]) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics.

[*Note 2*: For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on *A* forces prior side ef-

fects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on *A*. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — *end note*]

4 All modifications to a particular atomic object *M* occur in some particular total order, called the *modification order* of *M*.

[*Note 3*: There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads can observe modifications to different objects in inconsistent orders. — *end note*]

5 A *release sequence* headed by a release operation *A* on an atomic object *M* is a maximal contiguous sub-sequence of side effects in the modification order of *M*, where the first operation is *A*, and every subsequent operation is an atomic read-modify-write operation.

6 Certain library calls *synchronize with* other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store ([atomics.order]).

[*Note 4*: Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — *end note*]

[*Note 5*: The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release. — *end note*]

7 An evaluation *A carries a dependency* to an evaluation *B* if

(7.1) — the value of *A* is used as an operand of *B*, unless:

(7.1.1) — *B* is an invocation of any specialization of `std::kill_dependency` ([atomics.order]), or

(7.1.2) — *A* is the left operand of a built-in logical AND (`&&`, see [expr.log.and]) or logical OR (`||`, see [expr.log.or]) operator, or

(7.1.3) — *A* is the left operand of a conditional (`?:`, see [expr.cond]) operator, or

(7.1.4) — *A* is the left operand of the built-in comma (`,`) operator ([expr.comma]);

or

(7.2) — *A* writes a scalar object or bit-field *M*, *B* reads the value written by *A* from *M*, and *A* is sequenced before *B*, or

(7.3) — for some evaluation *X*, *A* carries a dependency to *X*, and *X* carries a dependency to *B*.

[*Note 6*: "Carries a dependency to" is a subset of "is sequenced before", and is similarly strictly intra-thread. — *end note*]

8 An evaluation *A* is *dependency-ordered before* an evaluation *B* if

(8.1) — *A* performs a release operation on an atomic object *M*, and, in another thread, *B* performs a consume operation on *M* and reads the value written by *A*, or

(8.2) — for some evaluation *X*, *A* is dependency-ordered before *X* and *X* carries a dependency to *B*.

[*Note 7*: The relation "is dependency-ordered before" is analogous to "synchronizes with", but uses release/consume in place of release/acquire. — *end note*]

9 An evaluation *A inter-thread happens before* an evaluation *B* if

(9.1) — *A* synchronizes with *B*, or

(9.2) — *A* is dependency-ordered before *B*, or

(9.3) — for some evaluation *X*

(9.3.1) — *A* synchronizes with *X* and *X* is sequenced before *B*, or

(9.3.2) — *A* is sequenced before *X* and *X* inter-thread happens before *B*, or

(9.3.3) — *A* inter-thread happens before *X* and *X* inter-thread happens before *B*.

[*Note 8*: The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with" and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before". — *end note*]

10 An evaluation *A* *happens before* an evaluation *B* (or, equivalently, *B* *happens after* *A*) if:

(10.1) — *A* is sequenced before *B*, or

(10.2) — *A* inter-thread happens before *B*.

The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation.

[*Note 9*: This cycle would otherwise be possible only through the use of consume operations. — *end note*]

11 An evaluation *A* *simply happens before* an evaluation *B* if either

(11.1) — *A* is sequenced before *B*, or

(11.2) — *A* synchronizes with *B*, or

(11.3) — *A* simply happens before *X* and *X* simply happens before *B*.

[*Note 10*: In the absence of consume operations, the happens before and simply happens before relations are identical. — *end note*]

12 An evaluation *A* *strongly happens before* an evaluation *D* if, either

(12.1) — *A* is sequenced before *D*, or

(12.2) — *A* synchronizes with *D*, and both *A* and *D* are sequentially consistent atomic operations ([atomics.order]), or

(12.3) — there are evaluations *B* and *C* such that *A* is sequenced before *B*, *B* simply happens before *C*, and *C* is sequenced before *D*, or

(12.4) — there is an evaluation *B* such that *A* strongly happens before *B*, and *B* strongly happens before *D*.

[*Note 11*: Informally, if *A* strongly happens before *B*, then *A* appears to be evaluated before *B* in all contexts. Strongly happens before excludes consume operations. — *end note*]

13 A *visible side effect* *A* on a scalar object or bit-field *M* with respect to a value computation *B* of *M* satisfies the conditions:

(13.1) — *A* happens before *B* and

(13.2) — there is no other side effect *X* to *M* such that *A* happens before *X* and *X* happens before *B*.

The value of a non-atomic scalar object or bit-field *M*, as determined by evaluation *B*, is the value stored by the visible side effect *A*.

[*Note 12*: If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then the behavior is either unspecified or undefined. — *end note*]

[*Note 13*: This states that operations on ordinary objects are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined below, and with

suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — *end note*]

14 The value of an atomic object *M*, as determined by evaluation *B*, is the value stored by some unspecified side effect *A* that modifies *M*, where *B* does not happen before *A*.

[*Note 14*: The set of such side effects is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below. — *end note*]

15 If an operation *A* that modifies an atomic object *M* happens before an operation *B* that modifies *M*, then *A* is earlier than *B* in the modification order of *M*.

[*Note 15*: This requirement is known as write-write coherence. — *end note*]

16 If a value computation *A* of an atomic object *M* happens before a value computation *B* of *M*, and *A* takes its value from a side effect *X* on *M*, then the value computed by *B* is either the value stored by *X* or the value stored by a side effect *Y* on *M*, where *Y* follows *X* in the modification order of *M*.

[*Note 16*: This requirement is known as read-read coherence. — *end note*]

17 If a value computation *A* of an atomic object *M* happens before an operation *B* that modifies *M*, then *A* takes its value from a side effect *X* on *M*, where *X* precedes *B* in the modification order of *M*.

[*Note 17*: This requirement is known as read-write coherence. — *end note*]

18 If a side effect *X* on an atomic object *M* happens before a value computation *B* of *M*, then the evaluation *B* takes its value from *X* or from a side effect *Y* that follows *X* in the modification order of *M*.

[*Note 18*: This requirement is known as write-read coherence. — *end note*]

19 [*Note 19*: The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

20 [*Note 20*: The value observed by a load of an atomic depends on the "happens before" relation, which depends on the values observed by loads of atomics. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here. — *end note*]

21 Two actions are *potentially concurrent* if

(21.1) — they are performed by different threads, or

(21.2) — they are unsequenced, at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation.

The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior.

[*Note 21*: It can be shown that programs that correctly use mutexes and `memory_order::seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result has undefined behavior. — *end note*]

22

Two accesses to the same object of type `volatile std::sig_atomic_t` do not result in a data race if both occur in the same thread, even if one or more occurs in a signal handler. For each signal handler invocation, evaluations performed by the thread invoking a signal handler can be divided into two groups *A* and *B*, such that no evaluations in *B* happen before evaluations in *A*, and the evaluations of such `volatile std::sig_atomic_t` objects take values as though all evaluations in *A* happened before the execution of the signal handler and the execution of the signal handler happened before all evaluations in *B*.

23 [*Note 22*: Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence rules. — *end note*]

24 [*Note 23*: Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the C++ program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. — *end note*]

6.9.2.3 Forward progress [intro.progress]

1 The implementation may assume that any thread will eventually do one of the following:

(1.1) — terminate,

(1.2) — make a call to a library I/O function,

(1.3) — perform an access through a volatile glvalue, or

(1.4) — perform a synchronization operation or an atomic operation.

[*Note 1*: This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — *end note*]

2 Executions of atomic functions that are either defined to be lock-free ([atomics.flag]) or indicated as lock-free ([atomics.lockfree]) are *lock-free executions*.

(2.1) — If there is only one thread that is not blocked ([defns.block]) in a standard library function, a lock-free execution in that thread shall complete.

[*Note 2*: Concurrently executing threads might prevent progress of a lock-free execution. For example, this situation can occur with load-locked store-conditional implementations. This property is sometimes termed obstruction-free. — *end note*]

(2.2) — When one or more lock-free executions run concurrently, at least one should complete.

[*Note 3*: It is difficult for some implementations to provide absolute guarantees to this effect, since repeated and particularly inopportune interference from other threads could prevent forward progress, e.g., by repeatedly stealing a cache line for unrelated purposes between load-locked and store-conditional instructions. For implementations that follow this recommendation and ensure that such effects cannot indefinitely delay progress under expected operating conditions, such anomalies can therefore safely be ignored by programmers. Outside this document, this property is sometimes termed lock-free. — *end note*]

3 During the execution of a thread of execution, each of the following is termed an *execution step*:

(3.1) — termination of the thread of execution,

(3.2) — performing an access through a volatile glvalue, or

(3.3) — completion of a call to a library I/O function, a synchronization operation, or an atomic operation.

[4] An invocation of a standard library function that blocks ([defns.block]) is considered to continuously execute execution steps while waiting for the condition that it blocks on to be satisfied.

[*Example 1*: A library I/O function that blocks until the I/O operation is complete can be considered to continuously check whether the operation is complete. Each such check consists of one or more execution steps, for example using observable behavior of the abstract machine. — *end example*]

[5] [*Note 4*: Because of this and the preceding requirement regarding what threads of execution have to perform eventually, it follows that no thread of execution can execute forever without an execution step occurring. — *end note*]

[6] A thread of execution *makes progress* when an execution step occurs or a lock-free execution does not complete because there are other concurrent threads that are not blocked in a standard library function (see above).

[7] For a thread of execution providing *concurrent forward progress guarantees*, the implementation ensures that the thread will eventually make progress for as long as it has not terminated.

[*Note 5*: This is required regardless of whether or not other threads of execution (if any) have been or are making progress. To eventually fulfill this requirement means that this will happen in an unspecified but finite amount of time. — *end note*]

[8] It is implementation-defined whether the implementation-created thread of execution that executes `main` ([basic.start.main]) and the threads of execution created by `std::thread` ([thread.thread.class]) or `std::jthread` ([thread.jthread.class]) provide concurrent forward progress guarantees. General-purpose implementations should provide these guarantees.

[9] For a thread of execution providing *parallel forward progress guarantees*, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step; once this thread has executed a step, it provides concurrent forward progress guarantees.

[10] [*Note 6*: This does not specify a requirement for when to start this thread of execution, which will typically be specified by the entity that creates this thread of execution. For example, a thread of execution that provides concurrent forward progress guarantees and executes tasks from a set of tasks in an arbitrary order, one after the other, satisfies the requirements of parallel forward progress for these tasks. — *end note*]

[11] For a thread of execution providing *weakly parallel forward progress guarantees*, the implementation does not ensure that the thread will eventually make progress.

[12] [*Note 7*: Threads of execution providing weakly parallel forward progress guarantees cannot be expected to make progress regardless of whether other threads make progress or not; however, blocking with forward progress guarantee delegation, as defined below, can be used to ensure that such threads of execution make progress eventually. — *end note*]

[13] Concurrent forward progress guarantees are stronger than parallel forward progress guarantees, which in turn are stronger than weakly parallel forward progress guarantees.

[*Note 8*: For example, some kinds of synchronization between threads of execution might only make progress if the respective threads of execution provide parallel forward progress guarantees, but will fail to make progress under weakly parallel guarantees. — *end note*]

[14] When a thread of execution *P* is specified to *block with forward progress guarantee delegation* on the completion of a set *S* of threads of execution, then throughout the whole time of *P* being blocked on *S*, the implementation shall ensure that the forward progress guarantees provided by at least one thread of execution in *S* is at least as strong as *P*'s forward progress guarantees.

[*Note 9*: It is unspecified which thread or threads of execution in *S* are chosen and for which number of execution steps. The strengthening is not permanent and not necessarily in place for the rest of the lifetime of the affected thread of execution. As long as *P* is blocked, the implementation has to eventually select and potentially strengthen a thread of execution in *S*. — *end note*]

Once a thread of execution in *S* terminates, it is removed from *S*. Once *S* is empty, *P* is unblocked.

15 [*Note 10*: A thread of execution *B* thus can temporarily provide an effectively stronger forward progress guarantee for a certain amount of time, due to a second thread of execution *A* being blocked on it with forward progress guarantee delegation. In turn, if *B* then blocks with forward progress guarantee delegation on *C*, this can also temporarily provide a stronger forward progress guarantee to *C*. — *end note*]

16 [*Note 11*: If all threads of execution in *S* finish executing (e.g., they terminate and do not use blocking synchronization incorrectly), then *P*'s execution of the operation that blocks with forward progress guarantee delegation will not result in *P*'s progress guarantee being effectively weakened. — *end note*]

17 [*Note 12*: This does not remove any constraints regarding blocking synchronization for threads of execution providing parallel or weakly parallel forward progress guarantees because the implementation is not required to strengthen a particular thread of execution whose too-weak progress guarantee is preventing overall progress. — *end note*]

18 An implementation should ensure that the last value (in modification order) assigned by an atomic or synchronization operation will become visible to all other threads in a finite period of time.