
4.5. Concurrency and Multithreading

Before C++11, there was no support for concurrency in the language and the C++ standard library, although implementations were free to give some guarantees. With C++11, this has changed. Both the core language and the library got improvements to support concurrent programming.

The following apply in the core language, for example:

- We now have a memory model, which guarantees that updates on two different objects used by two different threads are independent of each other. Before C++11, there was no guarantee that writing a `char` in one thread could not interfere with writing *another* `char` in another thread (see section "The memory model" in [*Stroustrup:C++0x*]).
- A new keyword, `thread_local`, was introduced for defining thread-specific variables and objects.

In the library, we got the following:

- Some guarantees regarding thread safety
- Supporting classes and functions for concurrency (starting and synchronizing multiple threads)

The supporting classes and functions are discussed in Chapter 18. The guarantees are discussed throughout the book. However, I want to give an overview of the general guarantees here.

The General Concurrency Guarantees of the C++ Standard Library

The general constraints the C++ standard library provides regarding concurrency and multithreading since C++11 are as follows:

- In general, sharing a library object by multiple threads — where at least one thread modifies the object — might result in undefined behavior. To quote the standard: "*Modifying an object of a standard library type that is shared between threads risks undefined behavior unless objects of that type are explicitly specified as being sharable without data races or the user supplies a locking mechanism.*"
- Especially during the construction of an object in one thread, using that object in another thread results in undefined behavior. Similarly, destructing an object in one thread while using it in another thread results in undefined behavior. Note that this applies even to objects that are provided for thread synchronization.

The most important places where concurrent access to library objects *is* supported are as follows:

- For STL containers (see Chapter 7) and container adapters (see Chapter 12), the following guarantees are given:
  - Concurrent read-only access is possible. This explicitly implies calling the nonconstant member functions `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `data()`, `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()`, and except for associative containers, operator `[]` as well as access by iterators, if they do not modify the containers.
  - Concurrent access to *different elements* of the same container is possible (except for class `vector<bool>`). Thus, different threads might concurrently read and/or write different elements of the same container. For example, each thread might process something and store the result in "its" element of a shared vector.
- For formatted input and output to a standard stream, which is synchronized with C I/O (see Section 15.14.1, page 845), concurrent access is possible, although it might result in interleaved characters. This by default applies to `std::cin`, `std::cout`, `std::cerr`. However, for string streams, file streams, or stream buffers, concurrent access results in undefined behavior.
- Concurrent calls of `atexit()` and `at_quick_exit()` (see Section 5.8.2, page 162) are synchronized. The same applies to functions that set or get the new, terminate, or unexpected handler (`set_new_handler()`, `set_unexpected()`, `set_terminate()` and the corresponding getters). Also, `getenv()` is synchronized.
- For all member functions of the default allocator (see Chapter 19) except destructors, concurrent access is synchronized.

Note also that the library guarantees that the C++ standard library has no "hidden" side effects that break concurrent access to different objects. Thus, the C++ standard library

- Does not access reachable objects other than those required for a specific operation,
- Is not allowed to internally introduce shared static objects without synchronization,
- Allows implementations to parallelize operations only if there are no visible side effects for the programmer. However, see Section 18.4.2, page 983.