NICHOLAS M. STOUGHTON

whither C++?

USENIX Standards Liaison

nick@usenix.org

AS I WROTE IN AN EARLIER ARTICLE, revision fever seems to be present in the various standards committees I work with. Along with the revision of POSIX and C (not to mention the possible revision of the Linux Standard Base), the C++ language standard is currently being revised.

To state that this is a big project would be a huge understatement. The working group itself is 3–4 times the size of any other comparable committee, and the scope of the work it has undertaken is on the order of 2^n. It appears a miracle will be needed if the end result is to obtain a clean, complete, and implementable specification in the timeframe they have set themselves.

Among the major things promised for the new version of the language (dubbed "C++0x," since it is hoped that the work will be complete in 2009) are:

- Concurrency support, including:
  - Concurrency memory model
  - Thread-local storage
  - Atomic operations
  - Thread support library
- Programmer Controlled Garbage Collection
- Concepts

The timetable for this project is also extremely aggressive. Officially, the registration ballot, which governs the outline of the document, started earlier this year and has only just completed as I submit this report. The working group members are busier than they have ever been, having held two plenary meetings so far this year, and with another to come in October. At the October meeting, the committee members must decide whether they are done writing the new standard yet or should take another year to complete it. Given the volume of work, it would be amazing if the document was ready to be voted out for final ballot in October, but that is the plan of record.

Concurrency Support

Concurrency support is a topic I've covered in this column in the past. With the advent of multicore chips in even cheap desktop systems, the need to correctly support multiple threads has never been more apparent. What isn't so clear is what should be in the standard once you say "support for concurrency" is to be included. There are plenty of existing threading libraries around, of which, in the C++ space at least, the Boost thread library is probably the most widely used. There are also many ap-

plications that use these libraries. What the language standard desperately needs to describe is how the memory model, the low-level atomic data types, and thread local storage work. It is not a major requirement, at least in my mind, to have yet another thread library *per se*. Such a library is proposed to include a new thread-launching API, thread cancellation, mutexes, condition variables, spin-locks, and the rest.

That would mean that all those existing applications would need to be ported to the new standard library. Multithreaded programming is hard enough at the best of times; once you have your program working, you are unlikely to want to rewrite it simply to use the new standardized threads, unless you are forced to (e.g., because you are using a third-party library that does).

One particular area where the current, nonstandard, threading libraries have problems is in handling thread cancellation. If a thread is blocked in an I/O system call (or any blocking system call, for that matter), it is sometimes convenient to signal that thread to tell it to abandon its wait and give up. POSIX has a pthread_cancel interface to do exactly that. Once the canceled thread acts on the cancellation request, it runs its cleanup handlers and dies. The thread cleanup handlers were designed with C++ exception handling in mind.

In the Rationale part of POSIX, thread cancellation is explained well:

> Many existing threads packages have facilities for canceling an operation or canceling a thread. These facilities are used for implementing user requests (such as the CANCEL button in a window-based application), for implementing OR parallelism (for example, telling the other threads to stop working once one thread has found a forced mate in a parallel chess program), or for implementing the ABORT mechanism in Ada.

> POSIX programs traditionally have used the signal mechanism combined with either longjmp() or polling to cancel operations. Many POSIX programmers have trouble using these facilities to solve their problems efficiently in a single-threaded process. With the introduction of threads, these solutions become even more difficult to use.

> The main issues with implementing a cancellation facility are specifying the operation to be canceled, cleanly releasing any resources allocated to that operation, controlling when the target notices that it has been canceled, and defining the interaction between asynchronous signals and cancellation. . . .

> *Cancellation Points*

> Cancellation points are points inside of certain functions where a thread has to act on any pending cancellation request when cancelability is enabled, if the function would block. As with checking for signals, operations need only check for pending cancellation requests when the operation is about to block indefinitely.

> The idea was considered of allowing implementations to define whether blocking calls such as read() should be cancellation points. It was decided that it would adversely affect the design of conforming applications if blocking calls were not cancellation points because threads could be left blocked in an uncancellable state.

> [from The Institute of Electrical & Electronics Engineers, Inc., and The Open Group, *Draft Standard for Information Technology—Portable Operating System Interface (POSIX®)*, 2007]

The current GCC model is to throw a special sort of exception when a thread is canceled. The exception is similar to typical C++ exceptions, except that it cannot be identified or ignored. The thread unwinds the stack, entering any catch(...) block and destroying objects as needed, until it exits, where it can be joined by another thread that is waiting. The exception is always automatically rethrown as necessary after any catch.

The C++ thread proposal wants to add its own thread-cancellation interface. The proposal at present, however, has nothing to do with thread cancellation as I have just described it (and as everyone is asking for). The proposed mechanism simply requests that the targeted thread throw an exception at the next point it notices the request to do so, and blocking system calls are not mentioned in the list of cancellation points. By calling this thread exception handling mechanism "cancellation," everyone is unhappy. It doesn't interrupt blocked system calls, and it doesn't terminate the thread. Any exception handler can "cancel the cancel" by simply catching and not rethrowing the exception. So it can't be implemented on top of pthread_cancel, and it does not serve any useful purpose beyond some sort of interthread communication mechanism.

But there is so much confusion caused by the terminology that it is proving very hard to come to consensus. There are those who believe that the C++ committee will be a laughing-stock in the community if it publishes a new revision of the standard and it does not have a thread API. I believe that it will be a laughing stock if it does, especially if the revision is anything like the one currently on the table.

There is also the question of mixed C and C++ applications to deal with in this case. Many applications use C language libraries, and there is no guarantee that C++ exceptions will correctly propagate up the stack through C stack frames. Exception-based thread cancellation may run into undefined behavior (e.g., core dumps) in a mixed-language environment. Most modern C compilers do have a mode to enable exception handling (for instance, gcc -fexceptions), but there is no guarantee that third-party libraries have been built this way.

This issue threatens to become a major stumbling block for the entire standard. It is possible that the standard may end up including a weaker than necessary thread library that is poorly designed, unimplementable on POSIX, and of little use to anyone. In fact, with this library included, the title of this article might be "Wither C++" instead of "Whither C++."

Garbage Collection

We've all used debugging aids such as Purify or Valgrind to track down memory leaks. In a well-formed C++ program, it is generally easier to avoid some of the more obvious memory leaks because of the way objects are destroyed as they go out of scope, and by use of the exception mechanism.

One thing that C++0x promises to bring to the table is the concept of Smart Pointers. Smart, or Shared, Pointers allow multiple pointers to the same object to exist, reference counting the object referred to via the smart pointer. This prevents the object from being prematurely destroyed, while ensuring that it does get destroyed once the last shared pointer goes out of scope.

However, this in itself doesn't cure all the problems of memory management. One group is trying to add explicit, programmer-controlled garbage collection to the language. There is a partially complete, experimental implementation of this, but little or no real programmer experience in using it.

It does look as if it might be a good debugging aid, to go with those we already have, but it is certainly unclear to many that this highly intrusive feature is worthwhile.

Every single object has a Garbage Collection (GC) attribute: it is either gc_required or gc_forbidden. Most objects shouldn't, in the end, care, in which case they should be gc_safe, which means that it doesn't matter if GC runs on this object or not.

A well-written program will gain nothing from GC (and in fact has to pay a small penalty for it). The real problems come with third-party libraries, which really should be entirely gc_safe. However, it isn't possible to have a program that has a mix of gc_required and gc_forbidden objects. So if a library chooses one of the required/forbidden attributes, the rest of the program (and any other library that is used) must go along with that.

Given the status of the implementation, and the fact that the "standardeze" has only just been written as I write this, it seems unlikely to me that GC will make it in. There remain a host of unanswered issues: Should we have finalizers as well as destructors? (A finalizer is run by the GC to release any nonmemory resources owned by the object.) Does this feature actually provide benefit? Will it lead to better or worse programs? One anecdote used recently related programming experience with GC leading to programs that "flushed the toilet when they noticed the ice-maker in the freezer was empty"; both are devices connected to the plumbing, but with no other obvious connection. Or, in programming terms, "I'm out of file descriptors; let's try garbage collection." GC is *entirely* about memory resources, and nothing else.

Good Stuff

OK, some of the new features in the language are definitely worth waiting for. The revision promises to give us variadic templates, r-value references, constant expressions, better support for generic programming (the "concept" idea currently implemented in ConceptGCC), better operator overloading, improved POD types (leading to better integration with C library functions), explicit virtual functions, strongly typed enums, and most of the library extensions from TR-1 (except the "Special Math" functions, which will move to a new, stand-alone International Standard of their own). If you want to see the details of any of these, go to http://www.open-std.org/jtc1/sc22/wg21/ and look at the individual papers.

POSIX and C++

On a somewhat separate note, several members of the Austin Group (responsible for the development and maintenance of POSIX), together with a good number of the C++ committee members, met independently and looked at the subject of providing a C++ language binding to POSIX. Other languages have done this in the past (notably Ada and Fortran), and it has been helpful. Until now, the answer to how to integrate POSIX facilities into a C++ program has been simply to use the C language libraries that POSIX specifies. However, there are numerous issues with this approach, and several places where POSIX meets C++ in ways that could be (and have been) implemented in a C++ library with C++ type semantics. Thread cancellation is an obvious candidate here, but the idea reaches across the entire POSIX range of functions, including such things as networking, file system access, dynamic libraries, and process control. Until this point, the Austin Group and C++ committee members have been operating as a Study Group under

the auspices of IEEE-PASC (the Portable Applications Standards Committee, the original authors of POSIX). This group has agreed to apply for a new IEEE project to develop such a language binding. The official name of the new project is the "POSIX/C++ Language Binding," but I'll refer to it as the "POSIX++" project.

Part of the scope of POSIX++ will be to define the interaction of the C language APIs and any C++ instantiation. For example, what is the interaction between iostreams and file descriptors? What happens if a C library does a pthread_cancel on a thread that was created in a C++ module?

It is also within the scope of POSIX++ to define new C APIs to allow for such mixed-language programs and to help C++ library developers implement the new C++0x library on a POSIX platform.

The POSIX++ project is just starting. If you are interested and want to be involved, please feel free to contact me.