What Happened to C++20 Contract Support?

Nathan Myers, 2019-08-04, rev. 1

The Summer 2019 meeting of the ISO SC22/WG21 C++ Standard committee, in Cologne, marked a first in the history of C++ standardization. This was the first time, in the (exactly) 30 years since ISO was first asked to form a Working Group to standardize C++, that the committee has removed from its Working Draft a major feature, for no expressible technical reason.

Background

C++ language features have been found obsolete, and been deprecated, several times, and actually retired slightly less often. This is normal as we discover new, better ways to do things. A feature may be recognized as a mistake after publication, as happened with `export` templates, `std::auto_ptr`, and `std::vector<bool>`. (The last remains in, to everyone's embarrassment.)

Occasionally, a feature has made its way into the Working Draft, and then problems were discovered in the design which led to removal before the draft was sent to ISO to publish. Most notable among these was Concepts, which was pulled from what became C++11. The feature has since had substantial rework, and the Standard Library was extended to use it. It is now scheduled to ship in C++20, ten years on.

Historic Event

One event at the 2019 Cologne meeting was unique in the history of WG21: a major language feature that had been voted into the Working Draft by a large margin, several meetings earlier, was removed for no expressible technical reason of any kind. This is remarkable, because ISO committees are generally expected to act according to clearly argued, objective, written reasons that member National Bodies can study and evaluate.

Nowadays, significant feature proposals are carefully studied by committees of experts in these ISO National Bodies, such as the US INCITS (formerly "ANSI"), and the British BSI, French AFNOR, German DIN, Polish PKN, and so on, before being admitted into the committee's Working Draft.

The reasons offered in support of adding the feature were, as always, written down, evaluated by the ISO National Bodies, and discussed. None of the facts or reasoning cited have since changed, nor have details of the feature itself. No new discoveries have surfaced to motivate a changed perception of the feature or its implications.

The feature in question, "Contracts", was to be a marquee item in C++20, alongside "Concepts", "Coroutines", and "Modules" – all firmly in, although Modules still generates questions.

Contract Support as a Language Feature

What is Contract support? It would have enabled annotating functions with predicates, expressed as C++ expressions, citing as many as practical of the requirements imposed on callers of the function, about argument values passed and the state of the program; and of details of results promised, both the value returned and the state of the program after. It was meant to displace the C macro-based `assert()`.

Uses for such annotations, particularly when visible in header files, are notably diverse. They include actually checking the predicates at runtime, before entry and after exit from each function, as an aid to testing; performing analyses on program text, to discover where any are inconsistent with other code, with one another, or with general rules of sound programming practice; and generating better machine code by presuming they are, as hoped (and, ideally, separately verified), true.

The actual, informal contract of any function – the list of facts that its correct operation depends on, and the promises it makes, whether implicit or written somewhere – is always much larger than can be expressed in C++ predicates. Even where a requirement can be expressed, it might not be worth expressing, or worth checking. Thus, every collection of such annotations is an exercise in engineering judgment, with the value the extra code yields balanced against the expense of maintaining more code (that can itself be wrong).

For an example, let us consider `std::binary_search`. It uses a number of comparisons about equal to the base-2 log of the number of elements in a sequence. Such a search depends, for correct operation, on several preconditions, such as that the comparison operation, when used on elements encountered, defines an acyclic ordering; that those elements are so ordered; and that the target, if present, is where it should be. It is usually asked that the whole sequence is in order, though the Standard stops just short of that.

Implicitly, the state of the program at entry must be well defined, and all the elements to be examined have been initialized. Absent those, nothing else can be assured, but there is no expressing those requirements as C++ predicates.

To verify that all the elements are in order, before searching, would take *n-1* comparisons, many more than *log n* for typical *n*, so checking during a search would exceed the time allowed for the search. But when testing a program, you might want to run checks that take longer, anyway. Or, you might check only the elements actually encountered during a search. That offers no assurance that, if no matching element is found, it is truly not present, but over the course of enough searches you might gain confidence that the ordering requirement was met.

Alternatively, the sequence may be verified incrementally, during construction, or once after, and the property exposed via its type. Or, a post-condition about each element's immediate neighbors after insertion may be taken, deductively,

to demonstrate the precondition, provided there were no other, less-disciplined changes.

An analysis tool, finding a sequence modified in a way that does not maintain its sorted order, might warn if it finds a binary search performed on the resulting sequence.

History

Contracts, as a feature, were first presented to the committee in 2012 as a proposal for a header defining a set of preprocessor macros, a sort of hypertrophied C-style `assert()`, practically useful only for runtime testing. This proposal bounced around until late 2014, when it was definitively rejected by the full committee, in effect pointedly inviting the authors not to bring it back. The committee did not want features that would increase dependence on preprocessor macros.

In the form removed from the Working Draft this year, contracts were a core-language feature usable for all the noted purposes. The feature was first presented in November 2014, initially just as a suggested direction of development. Over the following two years it was worked on by a small group to ensure it would serve all the intended uses. The result, as agreed upon within the group, was presented and voted into the Working Draft, essentially unchanged.

The central feature of the design accepted was that the predicate expressions were usable for any purpose, with no changes needed to source code from one use to the next. In any correct program, all are true, so their only effect would be on incorrect programs; at different times, we want different effects. To require different code for different uses would mean either changing the code, thus abandoning results from previous analyses, or repeating annotations, which would be hard to keep in sync. Or macros.

Almost immediately after the feature was voted in, one party to the original agreement – authors of the rejected 2012 design – began to post a bewildering variety of proposals for radical changes to the design, promoting them by encouraging confusion about consequences of the agreed-upon design.

One detail of the adopted design turned out to be particularly ripe for confusion. Recall that one use for contract annotations is to improve machine-code generation by presuming that what is required is, in fact, true. The text adopted into the Working Draft permitted a compiler to presume true any predicate that it was not generating code to test at runtime. Of course no compiler would actually perform such an optimization without permission from the user, but in the text of the Standard it is hard to make that clear. The Standard is tuned to define, clearly, what is a correct program, and what a compiler must do for one, but a program where a contract predicate is not true is, by definition, incorrect.

A lesser source of confusion concerned must happen if a predicate were found, at runtime, to be violated. Normally this would result in a backtrace report and immediate program termination. But it was clear that, sometimes, such as when

retrofitting existing and (apparently) correct code, the best course would be to report the violation and continue, so that more violations (or annotation errors) could be identified on the same run.

Choosing among these various behaviors would involve compiler command-line options, but the Standard is also not good at expressing such details. In the Draft, the choices were described in terms of "build modes", but many felt they would need much finer-grained control over what the compiler would do with annotations in their programs. Of course, actual compilers would support whatever users would need to control treatment of annotations, but at the time the only compilers that implemented the feature were still experimental.

None of the confusion was over which programs are correct, or what a correct program means, yet it exercised a curious fascination.

I do not mean to suggest that the design in the Draft was perfect. For example, as it was translated to formal wording in the Working Draft for the Standard, the effect of side effects in a predicate expression became "undefined behavior". It is obviously bad that adding checks to help improve and verify program correctness could so easily make a program, instead, undefined. This would have been fixed in the normal course of preparations to publish a new Standard, but it is notable that none of the proposals presented touched on this most glaring problem.

Similarly, it was clear that it would be helpful to allow marking an annotation with an identifier to make it easier to tell the compiler to treat it differently, but no proposal suggested that.

What Happened in Cologne

The profusion of change proposals continued in Cologne. Most proposals suggested making the feature more complex and harder to understand. The impression they created was of a feature that was unstable and unclear, even though they identified no actual problems with the version in the Draft.

The Fear, Uncertainty, and Doubt ("FUD") engendered by all the incompatible proposals predictably led members of the Evolution Working Group asked to consider them to look for a simpler version of the the feature to provide primitives that would be usable immediately, but that could be built upon in a future Standard with benefit of hindsight.

One of the proposals, not seen before the day it was presented, seemed to offer that simplicity, and the group seized upon it, voting for it by a margin of 3 to 1. It was opposed by four of the five participants of the original design group, because it was fatally flawed: in use, programmers would need to define preprocessor macros, and put calls to those in their code instead of the core-language syntax defined. It would breed "macro hell".

On top of its inherent flaws, it amounted to a radical redesign from what was originally accepted by the full committee. Making radical changes immediately

before sending a Draft Standard out for official comment was well beyond the charter of the Evolution Working Group at that meeting, which was expected to spend its time stabilizing the Draft. (The Chair of that group was well aware of this expectation. We are left to speculate why he permitted the vote.)

The immediate, predictable effect was panic. The most likely consequence of a radical change would be that, when asked for comment, some National Bodies would demand a return to the design they had originally voted in; others would demand the feature be removed, as evidently unstable. (There was never a practical possibility of sending the Draft out for comments with the voted change, or of a National Body demanding that version.) Such a conflict is among the worst possible outcomes in standardization efforts, as they threaten a long delay in publishing the next Standard.

Two days later, the same Evolution Working Group voted to remove the feature entirely. To head off a conflict between National Bodies, the authors of the original proposal and the authors of the change met and agreed to recommend that the committee accept removal. The following Saturday, the full committee voted for that removal, unanimously (with some abstentions).

What Happens Next

C++20 is now considered "feature complete". The Draft will be studied by all the interested National Body committees, which will come back with lists of changes that must be considered. (Changes they list generally cannot include features to be added.)

A new "study group", SG21, was formed to conduct formal meetings, with minutes, and produce a public document recommending action for a later standard. The target is intended to be the Standard after this, C++23, but since the Study Group has, as members, all the authors of all the proposals, to hope for agreement on a proposal in time for the next Standard would be absurdly optimistic. In particular, several of the members of the Study Group have explicitly denounced the central goals of the original design, in favor of preprocessor macros, so the group starts without so much as a coherent goal. All it has, really, is a feature name.

C++20 will be published with no Contract support.