Concepts in C++17

Andrew Sutton asutton@uakron.edu
Feb 12, 2012
Document: P0248

I propose to merge the Concepts TS into the International Standard for C++17. There are a number of compelling reasons to do this.

The design of the Concepts TS is mature. The syntax and semantics of the TS are rooted in the concepts design from 2003 and 2005 [1, 2, 3, 4], inspired by the work of Alex Stepanov [EoP], and with the active help of Alex Stepanov redefined through a complete specification of the STL algorithms (The Palo Alto report [5]). Note that only a single design issue against the TS was accepted during the 2015 Kona meeting.

An implementation of concepts is currently in GCC's trunk and has been used experimentally (as a branch) since 2013. It is a feature-complete implementation (modulo possible bugs, of course). These features will ship in GCC 6.0. I understand that initial work has also begun in Clang.

There is significant user experience. There have been early adopters and experimenters as long as there has been an implementation. Some users have shipped real products. Feedback from these users has been invaluable, but has not significantly affected the overall design of the language features in the TS. There are a number of libraries actively experimenting with concepts. Examples include my own Origin library, Casey Carter's implementation of the Ranges TS, Roland Bock's port of his `sqlpp` library, and Tom Honermann's Unicode `text_view` library.

Concepts have been used in educational settings demonstrating that they are easily taught, used, and liked by students that are not C++ experts for significant projects after just a couple of lectures of initial exposure.

Non-compiler issues reported thus far tend to fall into two categories: "I want to write template metaprograms with concepts" and "I want Haskell type classes". In the first case, concepts is not meant to eliminate all current metaprogramming techniques. We aim to make template programming and use easier by simplifying interface specification, not by offering a concept alternative to every metaprogramming implementation technique. In the second case, we already tried that. The current approach very purposefully goes in a fundamentally different direction, and does so without changing lookup rules or limiting what can be written in a template. We have done experiment to show that we could constrain template implementations as a compatible extension if we so desired. However, engineering the details would take time and what is currently available gives the users what they most ask for (better error messages, better/simpler notation, overloading, and better documentation).

My sense is that most people are comfortable, with the overall design. The issues above are not flaws in the design of concepts. Those are requests for different features.

Ideally, we would like to ship a concept-based standard library in C++17, but a TS (such as Ranges) might do. Users will not use concepts seriously until they can do so on their favorite platform. It

is unlikely that all major compiler implementers will ship concepts in 2016, but in 2017 they will be well on their way. We already know that concepts are useful for library design and implementation—even without a concept-enabled standard library. We have concept-enabled approximations of the standard library in active use. The current users of concepts would like better support, which is likely to appear soon with concept's inclusion into C++17. This pressure is likely to increase significantly after GCC 6.0 ships.

Without concepts supported in C++17, we risk a bifurcation of the C++ user community with many new libraries designed and/or implemented around concepts and others designed based on increasingly complicated uses of template metaprogramming to compensate for unconstrained templates. Separating the C++ library design community into a concepts school (relying on constrained templates offering precisely specified interfaces) and an unconstrained school (relying on compiler-time duck typing) would cause confusion. Some users will see that as yet another reason to abandon templates and generic programming in favor of other approaches, disrupting the progress towards safer, more general, and more efficient techniques. The best way to avoid this is to have concepts in the standard itself.

In conclusion, we must adopt concepts for C++17.

Bibliography

[1] B. Stroustrup, "Concept checking – A more abstract complement to type checking," Oct, 2003.

[2] B. Stroustrup and G. Dos Reis, "Concepts – Design choices for template argument checking," Oct, 2003.

[3] B. Stroustrup and G. Dos Reis, "Concepts – syntax and composition," Oct, 2003.

[4] G. D. Reis and B. Stroustrup, "Specifying C++ concepts," Apr, 2005.

[5] B. Stroustrup and A. Sutton, "A Concept Design for the STL," Jan, 2012.

[6] G. Dos Reis and B. Stroustrup, "Specifying C++ Concepts," in *Principles of Programming Languages (POPL'06)*, 2006.

[7] B. Stroustrup, "Concept checking – A more abstract complement to type checking," 2003.

[8] G. D. Reis, B. Stroustrup and A. Merideth, "Axioms: Semantics Aspects of C++ Concepts," Jun, 2009.