Concepts Are Ready

Gabriel Dos Reis
Microsoft

Background

Concepts are one of the most scrutinized proposed functionalities for C++. They are to support generic programming for the masses. They have seen several designs, the most recent being the "Concepts TS" published in October 2015 as an ISO C++14 Technical Specification. That specification was implemented and has been available in a GNU Compiler Collection (GCC) branch for many years, and has been shipping for production use as part of official GCC releases since GCC-6.0 – for almost a year now. That work is partially funded by the National Science Foundation, with the explicit purpose of bringing generic programming with C++ to mainstream.

Like any language feature with the potential of changing at scale how we write programs, how we think of programs, how we organize programs, concepts have attracted lot of opinions and suggestions over the years. However, judging from 20 years' participation in WG21 committee work and involvement in designing "concepts for C++" in an equal amount of time, they appear to me as the single proposed C++ feature that has been under a combination of extreme vetting, extensive reviews, protracted committee debates, with the ultimate effects of delaying the formal introduction of "concepts" in the main C++ standards. For example, one of the arguments against introducing "concepts" in C++17 was that there wasn't a single released C++ compiler at the time of the Spring 2016 meeting in Jacksonville. Yet, it was also clear that GCC-6.0 was shipping the very next month.

Another argument put forward at the Jacksonville meeting was that there wasn't enough "field user experience," yet we are now seeing proposed fundamental design changes to the "Concepts TS" (see P0587R0) with no evidence of "field user-experience" or C++14 or C++17 compiler implementation. A couple of the proposed changes are minor (e.g. replace "`concept bool`" with "`concept`") and are simplifications; they can be incorporated quickly if WG21 so chooses. Others are more substantive and are radical design changes (e.g. replace use of predicate operators with inheritance-style syntax) and aren't new ideas, we have seen them before in the failed C++0x concepts era. Yet, others like removing the abbreviated function declaration syntax gut the very basic design aim of making simple things simple. The ultimate effects of these proposed radical changes are more delays by way of fear ("*avoid concepts because WG21 might throw them out again*"), uncertainty ("*will my concepts-based code still compile tomorrow*"), doubt ("*should we still trust WG21 to serve the C++ community?*"). These delay, fear, uncertainty, and doubt are unlikely to bring in more "field user experience." Only more delays. This isn't to say that WG21 would be collectively, purposefully delaying "concepts". The observation is the combination of all these discussions, "new ideas", votes to not put concepts forward are having the practical effects of delays, and are causing confusion in the C++ developer community.

Introducing inheritance-style syntax in lieu of logical operators when combining concepts isn't just an awkward encoding of predicate composition, in the context of C++, forcing programmers *not* to express directly what they mean. It represents a radical departure of the Concepts TS design in the sense that it forces programmers to think "OO style" or concept hierarchies. For example, the paper P0587R0 suggests replacing

```
        template<typename T>
        concept bool Primitive = Integral<T> || Floating<T>;
```

with

```
        template<typename T>
        concept bool Primitive requires { /* … */ };

        template<typename T>
        extern concept Integral<T> : Primitive<T>;

        template<typename T>
        extern concept Floating<T> : Primitive<T>;
```

It must strike even the most junior language designer apprentice that this is a serious regression. Putting aside the obvious issue of verbosity and dumbfounding combination "extern concept", this proposed change appears to assume that the author of `Primitive<T>` has the rights or luxury of modifying the definition of `Integral<T>` and `Floating<T>`, or that such modification is even welcome. Furthermore, what happens when another programmer uses another concept based on `Floating<T>`:

```
        template<typename T>
        concept bool Fractional<T> = Floating<T> || QuotiendField<T>;
```

Is the programmer supposed to go back and modify `Floating<T>` or `Primitive<T>`? This is just one of the many examples illustrating the pitfall of trying to forcefully encode a natural predicate in an OO-style inheritance graph. It can be done in a closed world or in a nice academic paper about a "foundational concept calculus," but that isn't how you would design a language feature to be used by millions of C++ programmers in widely differing development environments. It does not support composition. It inverts logic, and direct, natural expression of ideas. It is not a good design.

Recommendation

The current design of "concepts" has been well tested, implemented, and used in production environments. While user experience suggests some syntactic simplifications might be welcome, there is no basis for radical design change. Stability in the core design is necessary to allow C++ implementers and C++ users alike to build confidence in WG21's ability to enact paradigm-shifting functionalities in C++, to solve problems affecting C++ programming. Concepts are ready. They are ready to be pulled into the C++ Working Draft. They have been lingering outside the main text for far too long.

Microsoft has begun implementing the Concepts TS. Like any other C++ toolset vendor, it will provide feedback to WG21 based on its customers' experience, and implementation experience as to what changes might be useful. At this point in time, based on the facts of GCC's implementation and its uses in the field, Microsoft's recommendation is: no more delay and uncertainly; bring the Concepts TS forward into the main C++ standards text. They are ready.