Doc. No. P1962R0
Date: 2019-11-18
Audience: EWG and LEWG
Reply to: Bjarne Stroustrup (bs@ms.com)

How can you be so certain?

Bjarne Stroustrup
www.stroustrup.com

I was reading a book by John McPhee [McPhee] and found this quote "Say not 'This is the truth' but 'So it seems to me to be as I now see the things I think I see.' " from David Love, one of the world's greatest field geologists [Love]. A bit convoluted, maybe, but it set me thinking. I have often wondered how some people could be as certain as they seem to be.

We need to think harder

When it comes to language (and library) extensions, we seem to spend more time on technical details (such as deciding which keywords are the least bad and exactly how to implement a feature) than on fundamental issues. We are far better technicians than we were in the early days of the committee, but I am concerned that our discussions of major changes are somewhat superficial. Sometime, I miss the rigor of a peer-reviewed academic paper – the papers at least partly based on empirical observations, that is. Often, I miss the careful weighing of pros and cons. Often, I wonder whether our weighing of evidence is thorough enough, consistent enough, and even if it pays attention to concerns that cut across the complete language and standard library.

- "I think so" is not a technical argument.
- "My company does it" is not a conclusive argument.
- "I badly need it" is not a conclusive argument.
- "All modern languages have it" is not a conclusive argument.
- "We can implement it" is a necessary but not sufficient reason to accept a feature.
- "Most people in the room liked it" is not a sufficient reason.

The last point may seem anti-democratic, but it is not. Which room? Were the people representative of the committee membership? Of the C++ community? What did they like? What were their reasons for liking it? Who was against? Why were they against? People in a room are typically self-selected and proponents of extensions are usually far more motivated and articulate than supporters of status-quo.

On a practical level, I think that simply adding up votes and seeing if there is a 3-1 majority is a mistake. We need such a majority and preferably more for consensus, but it is not sufficient, especially not when there are relatively few people in the room, when it's late in the week and people are tired, and when key and/or long-serving members are elsewhere. It is always worth considering who are strongly against and why. Neither strong proponents nor strong opponents are necessarily right. Sometimes strong emotions hide a lack of logical arguments.

We are defining a language for decades of use. A bit of humility is necessary.

Remember when

- it was popular to have all functions be members?
- virtual functions were so cool that many argued that every function should be virtual?
- public data was so "old school" that many argued that all data members should be protected?
- garbage collectors were deemed essential?

I suspect that a committee with the composition of today's committee would have followed these fashions. Today, we have different fashions, but it is still difficult to determine what's long-term useful and what's "just fashion." It is easy to get swept up in what is currently fashionable. Which problems are worth-while solving and how are both subject to fashions.

I am not arguing that we should slow down or speed up decisions. For example, I think we were far too slow for concepts and far too fast for <=>. I suggest that we could be more systematic, careful, and consistent in our reasoning.

Suggestions

It is easier to point to problems than to suggest remedies. There is no "magic source" for design that we could bottle. Also, language design for real-world use (which is what we are talking about) is not a simple predictable process. We never have complete agreement about exactly what "the problem" is, which problems need addressing, nor which of many remedies is best.

We should, however, try to learn from what worked well in the past and what caused lingering problems needing patch upon patch.

Different proposals require different kinds and different amounts of work.

I suggest we should look more at use patterns and interfaces, at least initially, and less on implementation details. Implementations tend to improve over years (and decades) provided the user interfaces and models of use are clean. We need to focus on "what?" and "why?" more than "how?" More on stable interfaces than on optimal performance given current techniques. A standard differs from most products by requiring stability over decades. I am of the impression that we have been focusing too much on detailed controls lately, on getting programmers to be very specific on "How?", thus making it harder to evolve use patterns and improve implementations.

Yes, the choice between detailed control and higher-level, more general interfaces is a design issue, rather than a decision-process issues, but choosing the more general interfaces allows implementation discussions to settle on an open set of alternatives, rather than pinning down every detail, thus simplifying standardization.

No language feature (or library) exists in isolation (or at least it shouldn't). Feature interaction is among the hardest of problems, and often underestimated both as a problem and as something useful. We should always consider interactions with other language features and standard-library components. That's hard, especially for other features that are still under development. This is one reason that after a major improvement of C++ we invariably need to do some minor cleanup and addition of minor supporting feature. We simply can't anticipate all that we will learn from deployment at scale. We should try not to elaborate features to try to anticipate every possible need. First, we can't. Second, if we try, we generate bloat that we have to live with "forever."

The committee process, "design by committee" (or more accurately "design by a federation of committees") in general tends to converge on designs that encompass the unions of all stated needs. For initial acceptance of a feature, that's bad. It's bloat. Not all stated needs are real and important for the majority of C++ programmers (directly or indirectly). We need to start out a feature minimal and then grow it based on feedback. "Minimal" should focus on what's fundamental, clean, and essential. Think of the original Unix (and it compare to modern derivatives). The growth from the initial nucleus of a feature to a full-blown facility should be generalization and integration with other language and library facilities. It should not be endless patches upon patches adding special cases. When such patching with special cases happens, it is a sign that the initial design was flawed. For example, I worry about last-minute "improvements."

There are two fundamental ways of approaching the unavoidable uncertainty in a design:

- add "improvements" until everybody feels well served
- cut until there is nothing left to cut and all there is left is principled and fundamental

After both approaches, experience will show the need for changes (pre-release) and additions (all we can do post-release). I clearly favor the second approach and consider it proper engineering based on principles and feedback. I consider the former hacking and politics. In a standards process, compromises are necessary, but we must try to ensure than compromises are not between irreconcilables or just union-of-features bloat.

When considering a problem, we should always try to articulate "who benefits?" "how do they benefit?" "how significant are the benefits?" For example, see [DiBello]. Also, "who might suffer new problems?" Be specific and concrete; "average users can't …" is neither. The benefits of a proposal are ***never*** obvious or self-evident to everyone. It is the task of the proposer to provide initial answers. Note that every proposal carries costs: committee time, implementation, documentation, teaching, dealing with older implementations and documentation, and more.

Setting goals is usually far harder than the detailed design and implementation of features. We are good technicians and we have theory and existing practice to guide us with the necessary work once goals have been established. Unfortunately, we are not good at agreeing on goals and articulating them. Often, we end up in a mess of requirements. Language design is not product development. We don't have a high management setting fundamental priorities for us. Few of us have workplace experience for that, if for no other reason that our firms are each in a specific business. I think product-development analogies have been seriously overdone over the last few years.

Neither arguments nor data are by themselves conclusive. We are far too good at fooling ourselves with clever arguments and data always require interpretation. One thing that the academic literature can be quite good at is the interpretation of data from experiments.

Our experiences are necessarily narrow and incomplete. The C++ world is too large and diverse for anyone to know all, and anyway, problems and styles of solutions change over time. Your perception of your company's or industry's needs may not be correct and even when they are, they may not be conclusive for the whole C++ community; serving your needs perfectly might even harm the C++ community as a whole. This is true for us all.

Many "perfect" languages have failed. If we are not careful, C++ can still fail. We need to be flexible and responsible. That is, our designs need to be such that they will still be relevant when the world changes (not "if the world changes"). We need to understand that every design is a risk because we can never be 100% certain that what we have correctly addressed important problems and don't block future improvements. However, that mustn't paralyze us. Doing nothing also has consequences (good and bad). We cannot avoid taking risks; let's make sure they are deliberate and considered risks.

We cannot make progress if we insist on perfection. We need to move cautiously forward basing initial designs on what has been seen to work and expect to have to elaborate later. This can be done only if we have a reasonably clear idea of where we want to go. In the absence of a view of a general direction, extension become mere hacking and we are doomed to apply patch after patch. A large working system is always the result of work on a smaller working system.

We never have a completely free choice in design and improvements. There are billions of lines of code "out there," millions of textbooks and popular blogs, and much old knowledge is stored away in millions of heads. Also, existing language features and the type system need to be respected for new and old facilities to interoperate smoothly.

Stability is a feature as well as a serious design constraint. It is always frustrating when old code and old design approaches stand in the way of "progress" but people ***really*** hate it when their code is broken. Some breakage is inevitable because much code depends on non-standard features or on features so obscure that they arguably always were bugs (in the code, in the compilers, or in the standard). I can summarize most people's attitude:

- "C++ is too complex. We need to make it smaller, simpler, and cleaner.
- And please add these two features.
- And ***whatever you do, don't break my code!***"

I think so too, but of course it's impossible. Even deprecating features has never really worked because users insist on compilers supporting old versions. Deprecating major features is impossible and deprecating a minor feature is an annoyance without significant benefits. C++ programs from 20 years ago still run; that's a strong argument for C++ use, partly because it gives hope that the code we write today will run 20 years from now.

Compatibility will always be a sensitive issue. I recommend focusing on coding guidelines and static analysis for achieving simplicity and correctness beyond what the language itself can guarantee. I consider the C++ Core Guidelines a good (though still incomplete) example of that [CG]. We can, and should, write type-safe and resource-safe C++. Language evolution should support this ideal (as documented in [D&E] and [Direction]). Applying techniques and new features to ensure type-safety and resource-safety to large existing code bases is very difficult, but still far more manageable than dealing with seriously incompatible versions of the language.

Implementation experience is usually beneficial for improving a design, but implementation work tends to freeze a design so that alternatives and improvements might be ignored. It is generally unwise to implement before the fundamental requirements and principles have been articulated (preferably in writing) and the key use cases have been chosen.

Usage experience reports are most valuable, but they are hard to get and impossible to get at scale. Usually, we have to make do with the experiences of a small group of self-selected, usually way-above-averagely-capable enthusiasts. Like implementation experience, early experience reports must be taken with a grain of salt, especially when they involve only a single, small, coherent group.

It is not just members of the community who wishes for "just two more features." We have a committee with 300+ members. It seems that essentially every member has a feature or two that they'd like to get into the language, and many have several. I have not changed my opinion that adding too many features could "sink C++." ***Remember the Vasa!*** [Vasa]. In fact, I think that the flood of new proposals has increased since I wrote [Vasa]. I think we are trying to do too much too fast. We can do much or do less fast. We cannot do both and maintain quality and coherence. We have to become more restrained and selective.

Every design has advantages, disadvantages, and limitations. We should never present a design without a serious and honest discussion of possible problems and alternatives. It is part of a proposer's job to examine problems; "pure sales jobs" are not intellectually honest. The joint "pro- and con-papers" on coroutines written by people from "opposing camps" were immensely useful ([Use][Impact]).

The ideal proposal reflects both some theory and some practical experience. We should also be more careful considering related (often academic) literature. For major proposals, there is always some related literature.

Design involves balancing different concerns and principles. There is never a single absolute and unbreakable principle that we can blindly follow. This is the reason that D&E lists a couple of dozen "rules of thumb."

Unfortunately, but inevitably, in the end much comes down to "taste" and crowds don't have taste. However, this note could be seen as containing a hidden check list – to be interpreted with taste.

Motivation

Recent discussions that prompted this note include:

- Major features: concepts, deterministic exceptions, and contracts.
- Minor features: uniform function call, spaceship, and "initializers in selection statements."

How can *you* be so certain?

Someone might reasonably point out: "You have often expressed strong opinions and sometimes even sounded angry when people didn't accept your arguments or proposals."

I should of course not show anger. Apologies. When it shows, it is typically the result of impatience after years of work or a feeling that not everybody or every argument are held to the same standard. Also, after years of work, it is hard to be patient with people for whom it is all new. Not only do I try not to show anger, I try not to feel anger, but I am not a saint and I care about these issues.

I am never 100% certain on the kind of topics we consider for standardization. I don't consider 100% certainty logically possible, and therefore accept the need for educated guesses. So far, we/C++ have not done too badly.

When I am pretty certain, my conviction is usually based on decades of prior art, theory, experience, and thought. For example:

- Constructor/destructor pairs and RAII: To many, those seemed to come out of nowhere in 1979, but they had deep roots in operating systems practice and reflected my inability to express those established practices directly in the languages of the time. I consider this the real foundation of C++.
- Concepts: I started looking at ways of specifying arguments for generic types and algorithms in the mid-1980s. Before that, I had found that the macro-based techniques we had used in the early years of C++ didn't scale and I followed the literature for more than a decade. The current design has its roots in work from 2003 (published about then) and even earlier. That work involved experimentation, implementation, academic publishing, committee papers, use, teaching, etc. I don't see significant improvements to users (as opposed to expert/implementer details) post 2015 or so. I consider concepts a necessary completion of the design of templates and an essential part of the support of generic programming.
- Exceptions: This issue is too inflamed to discuss in detail here, but the roots of the problem are in the multiplicity of error-reporting mechanisms. The discussions and research go back to at least 1974 (e.g., see [Errors]).
- Contracts: The use of invariants of various forms go back to Peter Naur's work in the mid-1970s. Much modern thought reflects Bertrand Meyer's work with Eiffel (both imitating it and opposing some aspects of it (e.g., catcalls and class invariants)). Other roots lie in decades of work on static analysis supported by assertions and interface specifications. The work involving "continuation", on the other hand, is relatively novel, backed mostly by logical arguments and experience at Bloomberg. Note that before the failure of the proposals for C++20, two previous attempts had also failed. We should consider why.
- Operator dot: One of my fundamental aims for C++ is (from the very start in 1979) to provide equally good support for user-defined and built-in types. For example, I'd like to be able to build an integer type that is as good as the built-in **int** in every way (with the possible exception of compile time). For types with public members, the lack of operator dot leaves a gap in how such types are controlled and accessed (e.g., I can't build a simple and general smart reference, a proxy). The very first extension proposal in 1980 was for an operator dot. C++ is not complete without **operator.()** or equivalent.
- Unified function call: The notational distinction between **x.f(y)** and **f(x,y)** comes from the flawed OO notion that there always is a single most important object for an operation. I made a mistake adopting that. It was a shallow understanding at the time (but ***extremely*** fashionable). Even then, I pointed to **sqrt(2)** and **x+y** as examples of problems caused by that view. With generic programming, the **x.f(y)** vs. **f(x,y)** distinction becomes a library design and usage issue (an inflexibility). With concepts, such problems get formalized. Again, the issues and solutions go back decades. Allowing virtual arguments for **f(x,y,z)** gives us multimethods.

All but contracts are part of my long-term view of what C++ should be. The static analysis part of contracts belongs there. I don't dislike the run-time checking part of contracts, I just can't claim them as part of my long-term view, e.g., they are not in D&E, though of course I knew about contracts then.

Conclusion

I cannot offer a simple clear conclusion. I don't think there can be a simple set of rules that anyone can follow to ensure a good design. This paper was motivated by the quote from a great empiricist urging intellectual humility. Elsewhere, he warned against theoretic/academic notions that had not been validated by observation of nature. In a sense that is the conclusion: don't be too certain of your facts and theories. Search out facts and work to develop a theory that match the facts.

Acknowledgements

Thanks to all who contributed in the reflector discussion of the draft of this paper.

References

- [McPhee] John McPhee: *Rising from the Plains*. The Noonday Press. 1986. Good reading if you appreciate good writing, geomorphology, the history of Wyoming, and moral dilemmas for scientists.
- [DiBella] Christopher Di Bella and JC van Winkel: Audience Tables. 2019-06-17.
- [CG] The Core C++ Guidelines .
- [Direction] H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong: *Direction for ISO C++*. P2000. 2019.
- [McPhee] John McPhee: *Rising from the Plains*. The Noonday Press. 1986. Good reading if you appreciate good writing and geomorphology.
- [Love] David Love, short bio.
- [Use] Geoffrey Romer, Gor Nishanov, Lewis Baker, Mihail Mihailov: Coroutines: Use-cases and Trade-offs . P1493. 2019.
- [Impact] Richard Smith, Daveed Vandevoorde, Geoffrey Romer, Gor Nishanov, Nathan Sidwell, Iain Sandoe, Lewis Baker: Coroutines: Language and Implementation Impact. P1492. 2019.
- [D&E] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.
- [Vasa] B. Stroustrup: Remember the Vasa! P0977. 2018.
- [Errors] B. Stroustrup: *C++ exceptions and alternatives*. P1947. 2019.