Doc. no.: P0976r0
Date: 2018-03-6
Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (bs@ms.com)

# The Evils of Paradigms

*Or*

*Beware of one-solution-fits-all thinking*

*Bjarne Stroustrup*

There is a notion (popularized by the American philosopher and historian Thomas Kuhn) of progress happening through "paradigm shifts" where an old (supposedly bad) way of doing or understanding something (a "paradigm") is replaced by a new (supposedly good) paradigm. Popular examples are the shift from Newtonian physics to Einstein's universe and the shift from geocentric view of the universe to the heliocentric. In programming, some people deem imperative, object-oriented, and functional programming different paradigms. I think the very notion of a paradigm does harm to use and to design because people all too easily fall into the trap of considering only one paradigm "good" and then try to fit everything into it, discarding all aspects of alternative "paradigms" as wrong or inferior (aka "If your only tool is a hammer, everything looks like a nail").

I reject that notion, as did Kristen Nygaard, who invented object-oriented programming. Instead, I see progress on a large scale as necessarily evolutionary, rather than revolutionary, progressing by older concepts, techniques, and tools being gradually absorbed into a more general framework. Consider:

- For a century after Copernicus, calculations of planet orbits were best done using the old geocentric model (cycles and epicycles!) because the heliocentric model had not reached the maturity needed to accurately model the sky. Even when a new way of looking at things is fundamentally better, it may not yet be sufficiently mature to be useful for practical tasks.
- Einstein's model of the universe is different from Newton's, and in important cases far more accurate. However, we spend almost all of our time in Newton's universe: Relativity is useful only for relatively esoteric topics, such as GPS implementation, HEP, and astronomy. To go shopping or to fly to Jacksonville from anywhere on earth, Newton's view is sufficiently accurate and much easier for us to deal with. Even when a new way of looking at things is fundamentally better, it may actually be inferior for simple, everyday tasks.

What does this have to do with C++? C++ is built on the idea of incremental growth and the gradual replacement of older facilities with newer ones where appropriate. Examples:

- A class is a generalization of a **struct** offering the opportunity for member functions and encapsulation. There were no attempts to ban **struct**s or to make every function a member function. There was no attempt to force all access to data to go through getters and setters. There was no attempt to guarantee complete independence of the type as presented to the user (the interface) from the representation type (the implementation). This was crucial for C++'s success. It allowed the efficiency and compactness necessary for systems programming. It allowed gradual adoption. It is still a major source of C++'s strength, as well as the root of some of its problems.

- A class is just a class until you start adding virtual functions and derived classes. There was no attempt to force everybody to fit every class into a hierarchy or to make every function a virtual member function. This may seem obvious today (as it did to me then), but the clamor for OO purity was dominant. The zero-overhead principle saved me (see D&E).

- Arrays (and especially the rules for array-to-pointer decay) is the root of much evil, but banning arrays was and is infeasible. We need arrays as the way of modeling the hardware notion of memory and for compatibility with billions of lines of code. The strategy adopted was to make arrays and pointers redundant in most code. We now have **std::vector** and **std::array**. Soon, I hope, we will have **std::span** (similar to what Dennis Ritchie wanted under the name "fat pointers"). In this direction, we also need **std::stack_array** to cover some important use cases. I don't know of any single abstraction that could completely replace built-in arrays for all uses. A general multidimensional array type with complete flexibility isn't all that hard to build – and "we" have known how to do so for about 50 years – but the cost of such generality was and is too high for C++.

- I intensely dislike macros, but macros are deeply ingrained in the C and C++ eco systems. Among other uses, they act as a plaster over language weaknesses. They allow us to fall back to token manipulation. That's what makes them poison to tool builders and consequently the root cause of the C++ community's serious weakness when it comes to tools. I didn't see a single facility for totally replacing macros while still remaining sufficiently flexible and efficient. Instead, I started an effort to gradually replace uses of macros with used of better-behaved features, such as **const**s, inline functions, name spaces, templates, initializer lists, **constexpr**. We are almost there! Static reflection might complete that effort (and feature macros could disrupt progress on tools). I know of no single language facility that could completely replace macros. I suspect that one that could would share many of the weaknesses of macros.

All of these decisions were controversial in their time; there were no shortage of people insisting on a clean break from the past, just as in the fashionable and academic languages. In my considered opinion, C++ would have been stillborn without these design decisions and the choice of gradual evolution rather than attempted revolution was essential. Thousands of languages made the opposite decisions, almost all died. It has been – and still is – essential that users can make significant improvements to their code without a complete and instant break from the past (modern examples are Perl5 vs. Perl6 and Python2 vs. Python3). Each failed or partially successful paradigm shift bifurcates the community.

In any constrained environment, a cleaner transition from the past to something new is often possible, but for the whole C++ community, no simple and general alternative exists. We couldn't achieve a clean

significant “paradigm shift” for everybody even if we had no compatibility constraints. There is no single general alternative to the current feature mix for everybody. Sometimes, we forget that and dream about the whole community simply adopting our favorite feature/technique/proposal/paradigm for everything. Even for an ideal alternative (which IMO, we will never have), that will not happen on a time-scale less than decades. It is fundamental for any evolutionary strategy that the language is viable at every point in time; we cannot ignore serious challenges for a longish period of time, waiting for a perfect solution.

Gradual transition and co-existence of older features with new also allows us to benefit from feedback in the evolutionary process. That’s good engineering as opposed to hopeful philosophizing.

In the context of design, one important and unfortunate side effect of paradigm-shift thinking in most of its variations is that a new facility must be complete. It must completely replace the old/bad facilities, so it must be able to do all that was considered good or useful from the bad old facility (or it wouldn’t be a new paradigm). This leads to feature bloat and delays as the new feature is extended and elaborated to meet the demand for completeness. The alternative to delays is often sloppy design and errors as it deprives the designers of the experience that an incomplete feature (while relying on the old for completeness and tricky examples) would give.

Gradual expansion, relying on feedback, is my ideal. Better an incomplete design than a poor/clumsy/bloated “complete solution.” We can always rely on the “bad old” facilities until we are confident that we have something genuinely better. Sometimes, we can even pull back from something that didn’t work out as expected (e.g., exception specifications). That said, I spend a long time thinking of the long-term aims and consequences. That’s necessary to avoid “gradual expansion” degenerating into opportunistic hacking. “Gradual expansion” does not mean “repeatedly replacing an old feature with one that better matches current fashion.” And “no”, this is not easy; there is no fool-proof formula. The thousands of failed language designs are witnesses to that.

Now consider current and future examples:

- For ages, we have had quests for the one-true-error-signaling mechanism. Currently, there are people arguing that all signaling of errors should be through error codes and others who would like to see all errors reported by throwing an exception. When designing exceptions, I came perilously close to recommending exceptions as a one-size-fits-all solution, but at least I didn’t recommend using exceptions for loop termination, as had been popular. The “exceptions are for exceptional events” and “use return codes if the direct caller can be expected to deal with an error” are decent rules of thumb, but do not cleanly and clearly help users select among the many alternative solutions (we should clarify). The real problem was and is that it is really hard to deal with a multiplicity of error-reporting mechanisms: **errno** (yuck!), returning a **struct**, a **std::pair**, a **std::optional**, a pointer that might be the **nullptr**, an **int** that might be -1, an “expected”, an out-parameter, or simple termination. Adding a mechanism to “solve the error-signaling problem” is more likely to add yet-another-alternative to this mess, than to solve it; as a user of N libraries, I now potentially have N+1 ways of signaling errors to deal with. Incidentally, this last point was a major motivation for introducing exceptions into C++; they handle a lot of the tricky cases elegantly (often through RAII).

- There are suggestions to extend module import to completely replace **#include** by allowing modules to export macros. I see no need for that: **#include** works as the low-level mechanism it was designed to be and interacts with the low-level macros in ways with which we have 45 years of experience. Macros are not modular: they don't obey the scope and type rules that are the basis of modularity. Furthermore **#include** and **import** co-exists rather nicely. You can use combinations of **#include**s and **import**s in user code and in the implementation of a module. There is no reason to cripple and complicate modules, lower the level of their implementation, and compromise compile-time performance by forcing them to deal with a mixture of macros and proper language facilities. If a library wants to "export macros" let it be represented to its users by a header file and **#include**d.

- Concepts were carefully designed with three notations for three different levels of complexity of use cases. The underlying principle was "Keep simple things simple!" aka "the onion principle" (like **for**-statements, where we have the fully general C-**for** plus the simpler, but simpler, range-**for**). However, there have been many suggestions to "simplify" by expanding ("bloating") each of the simpler notations to be able to express all cases (incl. cases of arbitrary complexity). This is both unnecessary (for example, the most verbose notation – an explicit **requires**-clause – can express all) and damages the simpler notations (especially the so-called natural notation) by eliminating the simplest and any most common cases in order to achieve uniformity and generality. We must not forget the simplest and most common cases: keep simple things simple!

Unfortunately, for major issues, "we know the problems with the existing solutions" is often used as an argument for untried alternatives. For C++, I preferred to use that argument against novelty where the problems seemed surmountable or where improvements to the older facilities seemed feasible.

Unfortunately, data necessary to resolve "paradigm choices" is hard to come by and available data is often ambiguous, biased, or hard to translate into concrete design choices.