*Bjarne Stroustrup is the designer and original implementer of C++ and the author of "The C++ Programming Language" and "The Design and Evolution of C++". His research interests include distributed systems, design, programming techniques, software development tools, and programming languages. He is actively involved in the ANSI/ISO standardization of C++.*
*Dr. Stroustrup is the College of Engineering Chair Professor in Computer Science at Texas A&M University. He retains a link with AT&T Labs – Research as an AT&T Fellow. Member of the National Academy of Engineering. ACM fellow. IEEE Fellow.* Bjarne Personal Page

**Part1: On learning and using of C++**

1. Why do you go to TAMU to teach programming? What's your favorite thing about the university? – xingranliuyun

> ➔ It was time for a change and I felt I had something to teach. I had friends at TAMU and it was one of the few universities that seemed serious about growing and improving. Some of the most enjoyable work has been designing and giving a first programming course to 1st year students.

2. Undeniably, the biggest problem of C++ is learning. Language-lawyer is a phenomenon that appears almost solely in C++. Many C++ programmers wasted so much time fighting against the language details, of which some are essential and some are unnecessary. C++ has many traps which we have to bear in mind before we can safely use it; C++ has so many pitfalls which leads to a tremendous amount of language tricks (some might call them "idioms" or "techniques"); they, together, lead to a very steep learning curve. Confronted with so many complexities, how can one learn and use C++ effectively from a practitioner's perspective? -- pongba&liujiang

> ➔ You may be accurate about the complexities of using C++, but then you may also be underestimating the trickery involved in learning and using other languages. Some problems only surface when the user population gets large and diverse. Also, C++ is used in a huge range of application areas and often the alternative to C++ is to learn several languages. Obviously, a professional should know several languages, but this should be taken into account when estimating difficulty/complexity. Also, if the complexity isn't in the language, it tends to be elsewhere, such as in the application code.

3. Of all the complexities in C++, some are essential, some are unnecessary (technical embarrassments, like you said). What're those, exactly? What's the right attitude towards them when learning or using the language? – pongba

> ➔ This is too big a question for a short answer, but I'm convinced that it would be possible to design a C++like language of maybe a tenth of the size of C++. The exercise would be non-trivial, though.
>
> Examples: both the syntax and the type rules are too irregular. There could be a

much simpler syntax and a far simpler and more regular type system, but C compatibility and C++98 compatibility would have to go out the window. Also, most of the default are the wrong way around or awkward: for example, constructors should be explicit by default, floating-point numbers should not be implicitly convertible to integers, and names should not by default be accessible from different translation units.

Attitude: Don't get stuck on details. Focus on becoming a professional (not a language laywer). Don't be too much in a hurry.

4. What's the first principle of learning C++? – pongba

➔ The way to master C++ is to focus on the fundamental concepts and techniques, rather than the language features – and in particular, not the minute details of language features.

5. What's the first principle of using C++? – pongba

➔ map ideas (concepts) directly into classes and templates

6. C++ is really hard to learn; normally one will have to read no less than 10 books to be adequately good at C++ programming. I really hope your new book is going to change the situation. – stevenmou

➔ So do I, and the feedback from people who have read drafts is encouraging. I think that one or two (well chosen) books will suffice for becoming a good C++ programmer. I have known people who did that. Don't think that knowing the most rules and buzzwords makes you the best programmer. A language is simply there for you to use to express ideas. Most of being a good programmer is clear thinking and application domain knowledge.

7. Nowadays, few systems are built with a single language. We tend to combine the power of different languages. In this case, C++ is often used to build parts of a system. How can we identify whether C++ is suitable for a specific module? Are there any fingerprints that could help us make decision on whether or not to use C++? -- Mike Meng

➔ C++'s main strengths are in flexibility and performance. If you need neither, use something else. One way of looking at a problem is to see if a set of classes – a small library – would help make the code cleaner, easier to get correct, and easier to maintain. If so, C++ may be the best choice for the design, implementation, and use of that little library.

8. I read your talk with Bill Venners in which you complained about the pervasive of

Object Oriented Programming. Does that mean that you decide not to support this specific style of programming? Generally, what kind of style in your opinion can be labeled as “good”? -- Mike Meng

> ➔ It’s hard to be specific. Most of my code is a combination of object-oriented, generic, and small free-standing classes. “Good” is a function of the quality of the match between ideas and code. Trying to define “good” in terms of language features or programming style fashions is a mistake.

**Part2: On future of C++**

1. What do you think of the prevalence of the “easy-learning” languages, do you think their population indicates the future trend? – kamala

> ➔ Maybe. The question is how many programmers there will be, what is considered programming, and what kind of professional background is considered reasonable for systems builders. I note that there may be more C++ programmers today than there has ever been (if not, it’s close). However, many tend to build infrastructure and is rarely heard from or noticed.

2. What’s the future direction of Generic Programming in C++ (especially after C++09)? Are there any new exciting things (besides those already in C++09) on the horizon? – longshanksmo

> ➔ I don’t know. Features such as concepts, auto, initializer lists, variadic templates, and rvalue references will revolutionize the way we express generic code. We will learn from that. I think that we’ll find that the integration of those features will be imperfect and lead to improvements.

3. C++, as a general-purpose programming language, is having its field squeezed by the more modern languages, where would C++ be in the foreseeable future, particularly in the concurrency age; would what used to be the unique advantage of C++ still be the unique advantage? – stlf

> ➔ C++ never had a unique advantage. It had strengths and weaknesses, as it has today and as all languages have. C++’s strengths include flexibility, performance, and co-existence with other languages (notably, C, Fortran, and assembler). If C++’s field really is being squeezed (and how would we know?) the reason will either be “marketing” against which the C++ community have few direct defenses or “because we have discovered to express our designs in frameworks that do not require flexibility and are efficient enough”. The latter is objectively a good thing. C++ is primarily a systems programming language and when an application area has matured to the point where “one standard way” will do, the systems programming language will be deployed elsewhere. I note a significant increase in the use of C++ in embedded systems programming.

4. C++ was and still is designed to be efficient; in order to achieve that, there're some non-trivial design trade-offs made to keep the abstraction penalty as low as possible (e.g. templates). However, they're not trade-offs without costs. For example, the static nature of templates makes it inflexible when it comes to runtime needs. In that case, Ruby's duck-typing seems to be a more natural implementation of generic-programming, although much less efficient. However, concurrent programming will definitely bring us a much heavier optimization means, in which case, would the compromise made in the design of C++ to keep the abstraction low still make sense, when people can resort to concurrency to gain efficiency? – pongba

> ➔ Ruby is often 50 times slower than C++. I don't think they are comparable languages. If you need performance in a scripting language, you implement your primitives in something like C++. If you need to do ad hoc programming at an application level where performance doesn't matter, you use a scripting language. It is a mistake to look for "the one true programming language."
>
> Also, I really appreciate strong static typing and design based on that for correctness. There is far too much talk about performance and far too little about correctness and structure.
>
> Besides, the individual processors are not getting any faster - in fact they are getting slower, being optimized for chip space and power consumption - so that low abstraction penalty could become **more** important for tasks that are not easily parallelized (and that is a lot of tasks).
>
> Remember: It costs thousands of instructions to start up a task on a processor and to get the result back, so there is a huge win in being able to execute in less instructions so that no spawning is needed. It's just like inlining vs. function call (the function call pre-and postamble costs). Also, more and more, we are going to measure speed in memory accesses rather than instructions - instructions are getting really cheap. Currently **every** multiprocessor/multicore is memory bandwidth limited - and also most single processors.

5. Are there any differences between the design principles of C++09 and that of C++98 (the ones in D&E)? Is there going to be a change in the design principles of C++1x? – liang

> ➔ Not really, at least from my perspective. My HOPL3 paper "Evolving a language in and for the real world: C++ 1991-2006." explains the principles and the evolution of C++ over the last 15 years. That's the place to look for a more detailed answer. It's available from my home pages.

6. You once said that there's a smaller, better language inside C++ dying to get out,

what's that language like, specifically? Is D it? – liang

➔ No. Nor is it Java or C#.

7. Money has always been C++ committee's biggest problem, how would the committee operate in the coming 10 years? – liang

➔ I don't know. Inertia will be a major problem. I fear it'll operate in just the same manner, and that wouldn't be good. However, I have reason to believe that it will move to a shorter release cycle (maybe 3 years). I don't see the money problems getting solved. People with money tend to prefer to spend them on proprietary solutions that they imagine will give them a competitive advantage – even if a joint effort would help everybody much more.

8. What responsibilities will you take in the future development of C++? -- liang

➔ Let's get C++0x finished before I think too hard about that.

9. What do you think of the place C++ is currently at in the Visual Studio language family? They have C# as the main static language, VB10&IronPython as dynamic languages, whereas C++ is the only choice when it comes to native code; Do you think that indicates C++'s place in the real-world development? Like Visual C++ Team said themselves: "We haven't forgotten C++"; does this represent the attitude to C++ in the industry? – liang

➔ C# is not a static language. It relies on the huge .Net framework. The amount of "native code" (read: C++) even within Microsoft is increasing and many of Microsoft's competitors (in a variety of business fields) do not want to depend completely on Microsoft, so they minimize their exposure to .Net. C++ is a far more portable language than anything you'll find in the .Net family (even C++/CLI). I think that the ones that do not will see business problems in the future as they become incapable of innovating in any way that doesn't fit .Net and is also done (in roughly the same way) by their competitors. To succeed in the long run, you need flexibility at many levels of your tool chain.

10. C++ is a language "designed by committee"; in which way does the nature of the C++ standard committee affect the evolution of C++, for good and for bad. – abware

➔ To the extent that C++ is designed by committee, it is bad and there are parts of the language that doesn't blend as well with other parts of the language as they should for that reason. What the committee seems good at is finding problems and limitation with individual features, but then improvements of individual features are not often as elegant as improvements to the interaction among features. This is definitely also true for interactions between library components and between

library components and language features. Again, read the HOPL3 paper.

11. How do you choose between adding new features to an existing language and inventing a new language? Sometimes, new features added to an existing language may look unnatural or difficult to use, but those features may be made more elegant if using a different syntax. -- WalterWalk

➔ You look at the problem. A successful language is successful because it addresses some problem better than the alternatives. Most new languages fail. I strongly prefer to start by building libraries.

12. What should be done in next generation of programming language from a researcher's view? – bipengace

➔ I don't know. I have never been a language researcher. Language research is almost always sterile. What works is using language-based tools to address interesting problems; sometimes the result is a new language or a new language feature. It is hard to know exactly how many languages are invented and die every year (for starters, how do you define "new language"), but a good estimate is about 2000 of each every decade – the survivors can be counted on a few hands. Have a look at my paper "A rationale for semantically enhanced library languages."

13. Is there any chance that, by proper "tailoring" (i.e. cutting out the obscure or normally uselessly complex features), we can provide a "smaller" C++, which is a subset of C++ and whose code can be compiled by any standard C++ compiler, and standardize this one, providing consistent ABIs and standard libraries. – cloudwu

➔ How do you decide what is "obscure and useless"? If you do it well, you can express it as a coding standard and enforce that. Unfortunately, my experience is that most coding standards are written by people who are inexperienced and fearful and do more harm than good by enforcing their fears. I think that the result of such "tailoring" should be (and if done well almost inevitably will be) domain specific. As an example see the JSF++ coding standard for safety critical embedded applications (i.e. airplane control). Link on my C++ page.