InfoQ's Video Interview with Bjarne Stroustrup

Q1. Could you introduce yourself to the audience please?

I'm the designer and original implementer of C++. I have served on the C++ standards committee from its start 30 years ago. I have been an industrial researcher at Bell Labs, a professor at Texas A&M and Columbia Universities, and now work for Morgan Stanley in New York City. I received Engineering's highest honor, the US National Academy's Draper Prize. I have written many popular and academic articles and several books. You can find many on my website, www.stroustrup.com . I still write code most days.

Q2.You were a faculty member during 2002 to 2014, and then joined Morgan Stanley in 2014. Why did you go back to the industry after a decade's career in academia?

By 2014, I had tried all the things you are supposed to do as a professor, teaching undergraduates, teaching graduates, graduating PhDs, writing academic papers, getting grants for research, sat on committees, etc. It was time for something new. Importantly, I felt I needed to get back to industry to learn what problems were current. In academia, there is a risk of solving irrelevant problems. Most importantly, I wanted to move closer to my grandchildren.

I didn't know the finance industry, but to my surprise Morgan Stanley not only had interesting problems related to distributed systems but also nice people working on them with C++. They also encouraged me to continue my work on improving C++. It also happens that Morgan Stanley's main offices were just 30 minutes from where my grandchildren live. I have now worked for Morgan Stanley for almost 6 years and still enjoy it. One interesting aspect is that Morgan Stanley has offices in many places around the world, so that I travel to get different perspectives on life and work.

Q3. What do you think are C++'s pros and cons compared with other emerging programming languages such as Rust, Swift, and Go?

I don't do language comparisons. To do that well takes serious work, and many would (unfairly) doubt my objectivity even if I had the time to make a serious study. However, I can outline some of C++'s strengths and weaknesses and you can decide how those relate to other languages and to your needs.

Strengths: C++ is great at delivering performance while managing complexity. It is stable over decades. It is not proprietary. It has good support for low-level concurrent programming and (type-safe) support for thread-and-lock level concurrency. It has multiple implementations. It runs on essentially all hardware. There is a large active developer community. There are a huge number of

standard and non-standard libraries. There are many conferences and user groups. For many problems, it is a complete solution. It is dominant or major in a huge number of industries. You can write type- and resource-safe C++ (use the C++ Core Guidelines).

Weaknesses: C++'s build systems and facilities for package management are not standardized. It is an older language with many older and unsafe features. It's complex. Much teaching of C++ reflects a bygone age. Proprietary languages and simpler languages usually have better tool support than C++.

Q4. Some developers think that C++ is becoming increasingly complex, which could be an issue. What's your view on that?

They are of course right: C++ is complex and becoming more so. So are other languages as they expand to address a broader set of problems. Java is an example of that – it has grown to more than three times its original size. We all want a simpler language, with just two more features, and no breakage of old code. That is of course impossible. What we have to look at is not the total complexity of the language but on how complex it is to write good code.

I try to evolve C++ towards a state where simple things are simple and cheap, and where more complex tasks are not unduly complicated or expensive. The simplest example is a for loop. The classical C/C++ for loop can do everything:

```
for (int i = 8; i<max; i+=2) do_something(v[i]);
```

including some bad errors:

```
for (int i = 0; i<max; ++j) do_something(v[i]);      // Oops!
```

However, most loops, simply traverse a data structure doing something with each element:

```
for (auto& x : v) do_something(x);
```

By adding the last construct (the range-for) C++ became more complex, but most programs became easier and safer to write. The old version is still there for when you need it.

Beyond the language features, we have the libraries. For example, the standard library now supports parallel algorithms

```
for_each(par_unseq, v, do_something);     // parallelize and vectorize
```

I refer to this as "the onion principle;" each time you peel a layer off the onion, you cry more. That is, the complexity and opportunities for errors increase.

There is much more that can be done, but you can solve problems easier, safer, and with better performance in C++20 than in earlier versions. For C++11, I observed that "C++11 feel like a new

language" because I could express my ideas so much better than in C++98. C++20 has a similar advantage over C++11.

Q5. What will be your advices and learning suggestions to the beginners of C++ programming language?

Don't try to learn it all at once. Don't try to learn C first. Don't think that C++ is fundamentally like Java or C#. Focus on fundamentals, key concepts, and techniques. Don't get obsessed with language-technical details. For every language construct and library, ask "Why is it here? What is it good for?" Exploring the "why?" questions leads to a deeper understanding of a language and saves time in the long run as compared to simply trying to remember all the rules.

When I was faced with the task of teaching programming to first-year engineering students, I wrote Programming: Principles and Practice using C++. That textbook incorporates my ideas of how to approach programming and C++. It also reflects years of trying out those ideas on students with a variety of interests and needs. It is now in its second edition, bringing the style of C++ up to C++14.

If you are already a programmer, you probably don't need the detailed approach of "PPP" (as my textbook is often called), but can go straight to A tour of C++. That is a short book (240 pages) giving the reader an overview of C++ and its standard library. It is now in its second edition covering C++17 and a few C++20 features.

Both books start with conventional imperative programming using a few standard-library components before proceeding to design of classes, error handling, and generic programming.

To learn C++, pick one of those books to start. Read it carefully and do a fair number of the exercises. Then proceed to try a real project, preferably under supervision of someone with patience and a solid development experience. Whenever possible, study with friends; it is much more pleasant as well as efficient to share the learning experience.

Look at the "C++ Core Guidelines" (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md). That's an ambitious open-source project to give concrete guidance on how to use C++ in an effective modern way. It is increasingly supported by static analysis tools and rely heavily on the ISO C++ standard library.

Q6. What do you think is the biggest challenge faced by C++? And how do you think it can be met and solved?

There are so many challenges. Not just for C++, but for all programming languages. For C++, I'd like to see

- de facto standard build systems and package managers
- teaching of C++ that reflects modern C++
- standard support for concurrency models that do not rely on explicit data sharing
- the C++ standard committee agree on directions for evolution
- simple facilities for static reflection
- functional-programming-style pattern matching
- static analysis guaranteeing complete type- and resource-safety to be available on all platforms
- a larger ISO standard library
- macro use eliminated

My most general aim for C++ is to be the best support for development of reliable, affordable, and efficient systems. My hope is that we are indeed evolving towards that.

Q7. As a programming language prevailing for over 20 years, C++ has stably been one of the top languages on TIOBE. What do you think makes the vitality of C++? Why do you think is the reason that C++ can last for decades?

That's a good question. C++ has been a major language for close to 30 years despite having no rich owner and no marketing organization. In addition, it is quite demanding of its developers and there is no shortage of competitors.

It is nice that Tiobe ranks C++ highly, but Tiobe is a poor measure of language use. It measures "noise" on the web. One enthusiastic student can "outweigh" ("out yell"?) hundreds of serious developers by simply posting frequently. It is hard to count programmers, but I'm pretty sure that Tiobe underestimates the number of C++ programmers. Professional surveys give different "rankings" (e.g., https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/), but yes, C++ is consistently among the top languages and the absolute number of C++ users keeps increasing (maybe by about 100,000 a year).

Stability is a feature. C++ is used for software that lasts for decades. We don't want such code to break or to need frequent major rewrites.

Essentially, C++ is the synthesis of two ideas:

- A direct map of language features to hardware: no implicitly added overheads in time or space. The ability to use the real machine (warts and all) rather than a virtual machine or a mathematical abstraction.
- Zero-overhead abstraction: the ability to define and use new types without overheads (time or space) that are not required by their desired functionality.

Many languages can do either, few – if any – can do both. So, C++ has a significant edge if you need both efficiency and abstraction; that is, if you need to exploit hardware well and also manage significant complexity.

The basic C++ language facilities are very general and efficient. They are not tied to a particular programming style. That's a strength because fashions in programming style change over time and facilities closely tied to a style, say object-oriented programming based on inheritance, can become a burden. C++ supports object-oriented very well, but it was never just an object-oriented language. Basic data abstraction and generic programming are very important today.

If I should point to just two features, it would be classes (with constructors and destructors) and templates (with concepts). Together they support flexibility, unsurpassed performance, and type- and resource safety.

Q8. Software is becoming increasingly important given the complexity of the world nowadays. For many developers, one programming language is far from enough to survive in the technology field, and application development tends to require multi-language programming, which further raised the bar for developers. Can you share some of your thoughts on this trend?

Yes, we need better developers, more professional developers, and we need to support them better with tools and rules that protect their professionalism. Not all developments require the same level of expertise. For foundational software, we need the rigor of professional engineering. For those, we need to raise the bar. Our civilization depends on that. For much other software, where lives and livelihoods are not at stake, we can do with less skilled developers.

A professional developer will know several programming languages, but the real key to quality software is an understanding of a complete tool chain (often involving several languages), an understanding of the application domain (including knowledge of the end-users), and the

fundamentals of computer science (such as data structures, algorithms, and machine architectures). I'm not saying that is easy, nor that every developer should know it all, but it is necessary that key developers live up to that standard. It is the ideal.

One reason I went back to industry after a decade in academia was to better appreciate the complexity of modern software development. I was worried that in academia I might underestimate real-world problems and might focus on the wrong problems.

Q9. How many programming languages in general do you recommend a developer or programmer to master? What will be the top 5 programming languages that you recommend to at least get familiar with?

I don't recommend specific languages and there is no "magic number" of languages everyone should know, but a serious developer should know several. C++ is a good choice if you ever have to deal directly with hardware or need to extract serious performance or low latency out of a complex system. You also need to know a scripting language and one of the managed languages. In addition, it is good to know a functional language. Today's languages are huge (especially when you consider their popular libraries and programming environments), so few people can be truly expert in 5 languages, but to be a professional developer you need to know one or two very well and the fundamentals of a few more. Over a career, the typical developer will seriously use many languages. It is dangerous to know only the latest fashionable language.

Q10. What are the skills that you think is important for every developer to master?

I can't think of specific skills that ***every*** developer need. It would be nice if everybody could approach problems logically and scientifically, but then parts of development of a large complex system (e.g., a game or an animation) require a more artistic approach. However, for most regular development tasks a logical mind and a sense of proportion is required for success. Those are skills that can be developed through mathematics, statistics, and probability theory.

For developers, the fundamentals of computer science are essential: some knowledge of data structures, algorithms, and machine architecture. Without those, a developer is stuck with doing derivative work and relying on others for performance and scale. This may be fine for some, but not for developers of foundational systems (e.g., drivers, libraries, communications systems, and networking). In addition, some understanding of security is needed for all who write software that interacts with other machines.

Everyone needs a good understanding of the tool chain that they are using, including programming languages, foundational libraries, and application-specific libraries.

In addition, a good developer is a good communicator. You need to listen to end-users and other developers to understand the problems. You need to communicate your ideas (in writing and verbally) to others, or they will have no impact. It's not all code.

Q11. Can you share some of your thoughts on the efficiency in programming? What do you think is the reason that some developers can be 10 times or even 100 times more efficient than others?

It is not easy to quantify "efficiency of programming." In particular, the person who writes the most code is rarely the person who is the most productive in the long run.

However, yes, I have seen people who consistently outperform their peers by 10 times or so and geniuses who occasionally gain a 100 times advantage. It is not just hard work – spending more hours – though that helps. You simply can't spend 10 times as much time as your "average" colleagues. There just isn't 600 hours in a week. A consistent focus over years, decades, can help. Each year's work building one the earlier ones, but I have met people whose performance I wouldn't be able to match if you gave me a 100 years to prepare.

General intelligence, solid education, deep understanding of the fundamentals, good understanding of tools, insights into the application domain, taste in which subjects to spend time on. That, and some people just seem to be different. The magnitude of differences is not all that different from what you see among musicians, linguist, mathematicians, chess masters, and soccer players. There is no "magic source" for exceptional performance that we can bottle.

Q12. What are your recommended programming tools in terms of operating systems, scripting languages, text editors, version control systems, shell, database engine, and some other tools that you feel absolutely necessary? Why do you like them?

I like Unix, of course, now in the form of Linux. I like Visual Studio as a development environment for C++. I use many systems, so I like tools that work on many systems. For example, my laptop is Windows, but most of the machines that do the "heavy lifting" at work are Linux. I use whatever toolset is available and whatever toolset is used by the people I work with. I try to be a light user of tools because when you are a heavy user, the many interacting tools often limits what you can do to the intersection of what your suppliers support. That can be a problem if you – like me – like

to experiment with new tools and techniques. For example, I have obviously to try out the latest versions of the various C++ implementations. I understand that my needs are quite different from many developers. I am rather keen on portability across platforms.

I can recommend the on-line compilers, such as the compiler explorer (https://godbolt.org/), for experimenting with the latest C++ features.

Q13. C++ 20 may be released in summer next year as planned, and it can be a major version. What are the new features added in C++ 20?

We will almost certainly deliver C++20 on time with the currently agreed-upon feature set. Furthermore, the major implementations will be feature complete before the ISO standard becomes official. The final technical vote should be in February (in Prague in the Czech Republic) and the ISO should issue the official document late in 2020, maybe October.

C++20 will include

- Modules – better code hygiene leading to faster compilations and (eventually) the elimination of slow and messy #**include**s and header files.
- Concepts – precise specification of a template's requirements on its parameters, simpler template syntax, more flexible and easier-to-use template functions. Generic programming becomes about as simple as "ordinary programming."
- Coroutines – Simple and fast generators and pipelines. Finally, after missing them for almost 30 years. They had been one of the reasons for C++'s early success.
- Ranges – Now we can write **sort(v)** rather than **sort(v.begin(),v.end())** without writing our own scaffolding.
- Dates – calendars and time zones. If you deal with business software, that's essential.
- Parallel algorithms – the easiest and safest way to benefit from multi-threading and multi-cores.
- Several improvements to concurrent programming.
- And many minor features.

I consider a feature major if it changes the way we think about code and software development.

Most of these features (all?) are shipping somewhere today. See https://en.cppreference.com/w/cpp/compiler_support. In general, cppreference.com is a good and reliable source of information.

Q14. What do you think can be highlighted of C++ 20? What are those C++ 20 features that will most interest developers?

I don't know. Developers often obsess over the strangest features and details, but I have a good idea what will help them most if they make the effort to learn to use the new features well.

My favorite feature is "concepts." Finally, after 30 years, I can precisely specify the requirements of a template on its arguments:

```
void sort(Sortable_range auto&);
```

where **Sortable_range** is defined as a range supporting random access of elements that can be swapped and compared using <:

```
template<typename R>
concept Sortable_range =
        random_access_range<R> && sortable(iterator_t<R>);
```

A "concept" is simply a compile-time predicate. It gives extreme flexibility because we can express first-order predicate logic.

Providing precise specification of arguments is almost exactly what I did when I introduced argument declarations for ordinary functions back in 1979:

```
double sqrt(double);
```

Before that, you could not specify function argument types in C. That is hard to imagine. I expect that soon it will be as hard to imagine C++ without concepts.

For many, modules will be the most important feature because it can significantly improve compile times. I have seen 2 to 50 times speedups! However, to get those dramatic improvements, you have to clean up your code so that it doesn't have surprising and undisciplined dependencies among its parts, and that is not always easy. Finally, we are able to get rid of **#include**s, header files, and most macros. As I said, that isn't trivial, but there are huge benefits to be had by doing so.

Q15. What's your expectations for C++ 20?

C++20 is going to be great! It has features that I have worked for and dreamed about for decades.

I expect fast adoption of many of the features. It was hard to move from C++98 to C++11, but we (the standards committee and the implementers) got better at compatibility, so the move from C++11 to C++14 and to C++17 was easier. I expect a move to C++20 to be easier still. However,

if you want to benefit from novel major feature, you do need to modify code. Just doing everything the old way will not give us the major benefits we'd like. For example, to gain major benefits from modules, you need to clean up your code organization. In particular, you need to eliminate unnecessary and dangerous uses of macros. Ironically, the messier your code it, the more benefits you can get, but of course that takes more work.

From my work with students, I know that using concepts in generic code makes the use of templates easier. From my experience with expert programmers, I know that concepts makes what used to be unmanageable reasonably easy.

For many, coroutines are new and to benefit you have to learn a whole new style. It is worth it, though, because it leads to simpler and faster concurrent programming. For example, here is a simple network echo service:

```
Task<> start()          // Infinite read/write socket task
{
        char data[1024];        // Buffer
        while (true) {
                auto n = co_await socket.async_read_some(buffer(data));
                co_await async_write(socket, buffer(data,n));
         }
}
```

Coroutines is already the backbone of many of Facebooks applications.

There are also many relatively minor improvements to concurrent and parallel code.

Q16. You've achieved so much in your area, how did you balance your work and life?

It is not always easy, but I think I have managed reasonably well. My children are doing well personally, professionally, and as good citizens. That's one key measure. When traveling for work, I often brought a child along to show them the world. When the children were young, we always ate our evening meal together (with no TV, computer, or phone), and always I read them "good night" stories. I spent endless time to take children to soccer, swimming, and music.

You cannot just work. That way, you eventually burn out. Life is not a sprint but a long-distance run. I have tried to live in nice places, and to visit nice places. My home town of Aarhus in Denmark, where I lived until I finished my MS degree, is a wonderful place. In general, Denmark is one of the nicest places on earth. Cambridge in England, where I got my PhD and now return to regularly, is a magical place. New Jersey, near Murray Hill where I worked for 24 years, is a good place to have a calm, productive, time and to bring up children. I always hated commuting as a complete waste of time, so I have tried to live 10 to 20 minutes from where I work.

Reading this, someone might get the idea that I come from a rich family. I do not. My family was thoroughly working class. All my uncles worked with their hands and I am the first in my family to attend high school. However, Denmark is a place that supports education and hard work. By some measures it is the most equal society on earth, and highly developed, so you don't have to be rich to do well and have a good life.

Q17. In addition to programming, what's your other hobbies and interests?

I read a lot, I listen to music, I like to visit interesting places, and to spend time with interesting people. I spend time with my family.

To get some exercise and fresh air, and to stress off, I run. I also like to walk to see places at a gentle pace. It is said that when Danes go abroad, their letters home are half about food. That seems right. A good meal with friends is a real pleasure.

I listen to classical music and rock, sometimes even when writing or programming. I read fiction and non-fiction – mystery, science-fiction, novels, a lot of history, and a bit of philosophy.

Maybe you could consider my work on the history of C++ a hobby; it doesn't directly relate to my day job. I have written a book ("The Design and Evolution of C++") and three papers for the ACM's History of Programming Languages conference. When the last of those is ready for HOPL'4 in London in 2020, I will have covered the first 40 years of C++. Thinking about history helps me consider why C++ succeeded and what weaknesses it has.

Q18. Not sure whether you heard about "996"? In China, it's quite common for a developer to have a lifestyle of "996" which means working from 9am to 9pm, 6 days a week. Can you share any thoughts on this?

I had not heard the "996" phrase, but I certainly know the phenomenon. For many, there are times in their lives where that's necessary. I have done that, repeatedly, but it is not a good strategy for every week for years. It's for bursts of activity at critical times. Add commuting time and there is nothing left. You get dull and possibly miserable. For the best work, we need time to rest and time to think, to reflect. For becoming a good person, we need time to spend with family and friends, to have interests outside work. I have seen people becoming successful in the workplace through "996", yet being miserable and unkind to people they interact with.

My ideal is a balanced life. For me, work is a major part of that, but certainly not the only one. For example, I have some of my best ideas with running, while resting, reflecting on "something else", and while chatting with friends.

Q19. How do you view the developer community in China? Especially C++ developers. Anything you'd like to tell them?

There are many good developers in China, but there are also many who use C++ in a 1990s style. That's suboptimal. My main advice is simple: get to use modern C++, that's C++11, C++14, or C++17. That way the code gets easier to write, easier to maintain, and runs faster. It pains me to see people suffer for lack of modern language support, using C++98 – or simply programming in a C++98s style. They are missing out on two decades of progress. It particularly pains me when people suffering in that way are students.

Part of my work has been to try to help developers to use C++ better. C++ is a very flexible, but also complex language. We need to think carefully about how best to apply it to real-world problems. Together with friends and colleagues, I have developed a set of recommendations for effective use of modern C++: The C++ Core Guidelines (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md). It's an open-source project with many contributors. The document outlines the overall philosophy and lists many practical rules. Most of the practical rules can be checked statically, so that eventually we will have tools that finds and help remedy violations. Some static analyzers relating to memory safety and other key topic are already shipping in Visual Studio. I expect that such analyzers will eventually be available for all C++ implementations.
Using such guidelines and analyzers, a novice developer can avoid novice mistakes and we can find "bad code" (meaning old-fashioned, unsafe, and inefficient code) in older code bases. There is some minimal library support ("The Guidelines Support Library), but mostly the Guidelines simply relies on the ISO C++ standard. Some of the Guidelines Support Library facilities are accepted into C++20.

Part of the guidelines come from my books, "The C++ Programming Language (4th Edition)" and "A Tour of C++ (Second edition)". "The tour" actually refers to the guidelines. It is a thin book (240 pages) aimed at bringing developers who are already competent programmers up to speed with C++17 and parts of C++20. Both books are now available in Chinese as well as English.

I think the most important advice would be: Don't get lost in the details. Focus on the fundamental concepts and techniques. Dig into details only as needed. The Core Guidelines and "The Tour" can help with that. For language rules and standard libraries, I recommend the online cppreference.com . The C++ Foundation's website isocpp.org has a lot of good information and a steady stream of news about C++. There are a lot of videos on C++ on the web, but unfortunately for many people in China, most are on youTube.

I suggest that developers join (or start) C++ users' groups so that they can share experiences and useful information with each other.