C++20 – iX Developer: Answers from Bjarne

“C++ has support for that”

C++ is already over 40 years old: Why is it still worthwhile learning this programming language for developers today?

The fact that C++ is 40 years old and has an active user community of something like 4.5 million developers is a strong indication that it is worth learning. No language lives that long or grows a user community that large without some serious strengths. Most languages die young or get stuck in suspended animation.

Independently of that, C++ has strengths when it comes to learning a range of useful concepts and techniques. If you want to know your system at the machine level (bits, bytes, words, and addresses), C++ has support for that. If you want to understand the use of static (compile-time) types, C++ has support for that. If you want to organize your program by defining and using your own types, C++ has support for that. If you want to manage general resources (e.g., memory, files, locks), C++ has support for that. If you want to tune your code for optimal performance, C++ has support for that.

Another way of looking at this is to note that C++ supports conventional C-style programming, data abstraction, object-oriented programming, generic programming, and concurrent programming.

Yet another way of looking at that is that C++ is a backbone of major applications, in embedded systems, games, finance, bio-tech, medicine, aerospace, automotive, AI/ML, and much more. Many of the world’s most prominent tech companies use C++ for their key systems.

That said, a professional developer knows more than one language, and C++ is a good choice as one of those.

“Think of C++ as of learning a musical instrument”

Is there a piece of advice you can give to C++ newbies for getting into this challenging language?

Concentrate on the fundamentals, on the concepts, principles, and techniques. Try not to get lost in the many obscure technical details. Don’t try to understand everything. You simply can’t understand everything immediately. Think of the mastery of C++, its techniques, libraries, and tools as you would think of learning a new natural language well enough to hold a serious conversation in it or playing a musical instrument well enough for someone genuinely wanting to listen to your performance.

To help, I have written two books:

- “Programming: Principles and Practice using C++” for people who have not programmed before. As its title indicates, it uses C++ as a vehicle to teach the fundamentals of writing good programs.
- “A Tour of C++” for people who already know a fair bit about programming from some other language or from an older version of C++. It is a brief (200 page) overview

of the basic facilities offered by C++ assuming that you have the maturity to search out further material when needed.

Both books are in their second edition and both are available in German as well as several other languages.

Writing good programs takes some perseverance and some understanding of the application domain and the tool chains involved. Often, the programming language is nowhere near the most complicated and tricky subject to be mastered to produce something useful.

When done well, programming – software development – is a noble art as well as a very useful one. It is worth investing significant effort to master. Think of it like Mathematics or Physics, not like a low-level skill to pick up in a couple of weeks. C++ can be a key part of such a quest for mastery.

Lack of women in Computing and the C++ community

C++ is still being perceived as a predominantly "male domain" by some: What do you think?

That's very sad! It's even worse because in many places it's not just perception, it's true. It shouldn't be. Worse still, it's not just C++, it's huge tracts of IT/CS. We have to do something about that, but it has proven surprisingly hard.

How would you encourage more female developers to dive in?

That's obviously harder than it sounds, or we would have solved the problem decades ago. There are undoubtedly many causes of this sorry state, but from where I sit, it seems to a large extent to be a function of perception, rather than anything hostile or offensive done by Computer Science and IT people. Undoubtedly, not everyone in IT/CS/C++ is perfect, but on average we can't be worse than in so many fields where women have succeeded in large numbers, such as medicine.

One problem is that "the media" often describes computing as a field of lonely, antisocial men constructing harmful systems. Journalists and family members often advise young women to avoid "male professions." The massive positive contribution of computing in general and C++ in particular to modern society is often forgotten and the emphasis is placed on harmful practices, such as hacking, and on failures. The fact that computing is often a social activity involving groups of people working together to solve important problems in education, medicine, science, and everyday business is forgotten. I dread the picture of the software developer as a fat, ill-dressed, slob, sitting alone in the dark surrounded by empty pizza boxes cranking out incomprehensible text. That horror is not representative.

Note that I repeat "often". Obviously, not every description perpetuates the negative images, but enough do to do harm. We see heroic doctors saving lives, not the team of engineers and software developers who design, implement, and maintain the doctors' essential tools and the medical research that those tools and techniques build on.

Computing needs more emphasis at Schools

I feel that too little of the teaching in schools emphasize the value to society of computing and the essential role of developers. The emphasis too often is on making everything easy and fun, on instant gratification of the students. In many places, there is a tendency to avoid difficult topics to keep

enrollment high and to keep students happy. For example, to teach subjects that very early let the students do graphics, image manipulation, and animation without first (or ever) giving the students the information needed to understand exactly what's going on. I have seen that both for high-school students and engineering students. An analogy: it's OK not to understand the combustion engine if all you want to do is to drive a car, but an engineer needs to know the basics of how an engine functions.

That approach puts C++ at a disadvantage compared to languages designed for ease of use as opposed to system building. I wish schools would treat computing as a serious topic the way they do Math, Physics, and Biology – at least for students aiming to become professionals in fields such as computer science and engineering

Why are men less put off by the negatives than women? I don't know. I guess that part of the answer is that rockets and games traditionally appeal more to men and that parts of society discourage women from entering traditionally male fields. Part of the reason is simply that there are too few women in computing; this is a self-perpetuating problem. In many countries, we failed to get a critical mass of women into computing while computing was young. People manage the challenges of education far better if they are part of a group of friends, and looking out over an auditorium of 250 students and seeing just 20 women, I see a major problem. I understand why proportionally more women dropped out than men, despite objectively doing relatively better on the material taught. Actually, my graduate classes at Columbia University have about a third women, so maybe we are seeing improvements. I am still optimistic.

C++ is a small part of this picture, but I fear that C++ that C++'s image on "close to the hardware" code, embedded systems, engineering, games, finance, and large systems make the problems worse.

Make society in general, and women in particular, understand how much good is being done by good, professional software development and the problem will eventually be solved. For a young person to devote many years to master a field, first that field must be perceived as worth-while, possibly even noble.

That done, the emphasis must be to make it attractive for women to stay in the field. The life/work balance problems can be harder for women. There has to be opportunities for socialization that doesn't drift into traditional male activities. There has to be ample support for pregnancies, maternity leave, and child care. Some societies are far better than others at that.

I don't have any simple explanations and answers. I have undoubtedly failed to understand or mention something crucial. Like most people, I find it easier to point to problems than to recommend answers, but we have to keep trying. I'm optimistic that eventually we will see many more women in computing and C++, and that they will be happy to be there. The field will be better for it.

C++20 related: New Features

Which new features will have the most impact on C++ in the long run, and why?

Modules, concepts, and coroutines.

Modules will finally bring us better code hygiene (less macro pollution), easier composition of code out of libraries, and much faster compilation. I am looking forward to mature module

implementations that integrate well with the program development environment in general. That will take time. Please remember that C++20 won't become official for another few months after the ISO secretariat in Geneva finishes its non-technical review, so we should be happy with early implementations and not be too impatient with imperfections. We also need to learn to use modules well; much of what we do are infected with bad habits of almost 50 years of #include and #define. Just because the preprocessor is familiar doesn't mean that it is simple to use well.

Concepts will eventually make generic programming very similar to what today is ordinary programming. This will make our code far more flexible, reusable, and efficient. When I designed templates, I wanted three things:

- Expressiveness Generality
- Unmatched efficiency
- Precise specification (type checking)

I got the first two, but not the third. That was enough to make generic programming and template metaprogramming run-away successes, but it strained the language facilities and their implementations beyond reasonable limits. The fact that reasonable people were willing to suffer the imperfections to gain the benefits is a strong argument for the fundamental ideas being sound, so I and friends set out to provide the third item, the specification of a template's requirements on its arguments. Alexander Stepanov, the father of C++-style generic programming, named such specifications "concepts." They are basically just first-order predicate logic, making them simple to use and to think about while still extremely flexible. They carry no run-time cost. Over the next few years, we will see concepts slowly replacing types and class hierarchies as the primary way of specifying function arguments. Eventually, we will not understand how we could ever have done without them. It will be like today looking back on pre-prototype C code.

Coroutines will free us from explicitly saving state functions where the next value depends on previous calls. Such computations are very common in concurrent computations and I expect coroutines to dramatically simplify the way we organize many concurrent systems. With simplicity comes efficiency in time and space. That's important because some concurrent systems serve hundreds of thousands of tasks. Coroutines were parts of the earliest C++. In fact, the first C++ library was a coroutine library with some support for Simula-style simulation. It was essential for the early adoption and spread of C++. If not for coroutines, we wouldn't have a modern C++ to discuss. Unfortunately, we lost them in early 1990 when more complex machine architectures made it harder to implement coroutines well.

These three features are mentioned in my "The Design and Evolution of C++" from 1994 and have been part of my dreams for C++ from the start. I'm very happy to have been one of the people making those dreams come through. These three features are deeply ingrained in what C++ is, not just additions on top of earlier facilities. They relate to code organization, the type system, and to the use of hardware resources.

To learn more of the aims, design, and evolution of C++, I recommend my three papers for ACM SIGPLAN's "History Of Programming Languages" conferences (HOPL-2, HOPL-3, and HOPL-4), and especially the latest: Thriving in a crowded and changing world: C++ 2006–2020. It's a huge topic, so that paper is not a quick read.

Few barriers to switch to C++20

In your opinion, what could be possible barriers to switching to C++20?

There are very few barriers to switch from C++11, C++14, or C++17 to C++20. The language, standard libraries, and the major implementation are very compatible. But do remember that C++20 isn't yet official, so anyone who used the C++20 switches on the implementations before those implementations promises that C++20 is fully supported are by definition on the bleeding edge. In practical terms, using the new features well will involve some learning and reorganization of code, especially for modules.

Which innovations have not (yet) made it into the current standard, and why?

C++ grows based on feedback from use and a developing consensus in the ISO standards committee. Nobody gets exactly what they want and facilities take time to mature. Every new standard simply marks a place in the evolution of C++ towards becoming a better tool for real-world software development. My personal aims include

- Making simple things simple (without making complicated things impossible or unreasonably hard to do)
- Using hardware well
- Type- and resource-safety
- Zero-overhead abstraction

I favor facilities – small or large, language or library – based on how well they support those aims and on how much they help users relative to the effort of introducing them. You can read about such ideas in the committee's Direction Group's regularly updated writeups. H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong: DIRECTION FOR ISO C++ . Among many other topics, that paper has lists of suggested improvements.

I look for major features that makes something fundamentally simpler, such as a more complete model of concurrency (executors), static reflection, and functional-programming-style pattern matching.

I look for library components that are either foundational or give crucial support for a specific kind of application, such as units, sound, and simple graphics.

I look for small features that simplify notation or save programmers from having to write simple commonly useful functions and classes.

I do not favor a multitude of minor convenience features with limited applicability. I see a steady stream of proposals for small features that we have lived happily without for 40 years. Each would add just a little bit of complexity to C++, but in total, they would make a terrible mess. I wrote a paper about that: Remember the Vasa! . From a 1992 paper:

> We often remind ourselves of the good ship Vasa. It was to be the pride of the Swedish navy and was built to be the biggest and most beautiful battleship ever. Unfortunately, to accommodate enough statues and guns it underwent major redesigns and extension during construction. The result was that it only made it half

```
way across Stockholm harbor before a gust of wind blew it
over and it sank killing about 50 people. It has been
raised and you can now see it in a museum in Stockholm.
It is a beauty to behold - far more beautiful at the time
than its unextended first design and far more beautiful
today than if it had suffered the usual fate of a 17th
century battle ship -- but that is no consolation to its
designer, builders, and intended users.
```

Future: "We have to be more patient."

What can developers look forward to in C++23?

We have to be more patient. Covid-19 slows down the committee work as is does much other work. I hope for bug fixes, a few minor features "to complete C++20" and possibly one or more of

- Standard library modules
- Library support for coroutines
- Library support for networking
- Executors
- Functional-programming-style pattern matching
- Static reflection

from the committee's pre-Covid-19 plan for C++23.

What are your personal wishes for the future – not only, but also for the development of C++?

Like most people, I have a long list of what I'd like to see improved in the world, but I'd rather not get into politics beyond hoping for a safe, effective, and cheap Covid-19 vaccine and that people will get their act together about climate change.

Now back to topics where I actually have some insights and possibly even some influence. For C++, I want guaranteed type and resource safe code. I'm part of a project to give us that. If you can write arbitrarily complex C++ code, you cannot have such guarantees. However, we can write rules for the use of C++ that can be checked by a static analyzer to guarantee that. The project is The C++ Core Guidelines encouraging a modern style of C++ that we can check to eliminate memory corruption, memory leaks, and type violations without introducing a garbage collector or imposing costly run-time checks.

It is based on the subset-of-superset idea: first you extend the language with foundational libraries and, that done, you can ban features that cause trouble. That approach is necessary because the dangerous features are often exactly the ones you need to build the foundational libraries. Static checking is needed to detect violations of the rules that might lead to errors. Currently, Visual Studio C++ ships with a checker and Clang tidy has a few checks. My hope is that something like those rules will become very widespread and that checkers will become available by default in all major C++ implementations. I hope for that to happen soon, it would eliminate so many widespread and unnecessary problems.