Dr Bjarne Stroustrup: An Interview

Emyr Williams begins a new series of interviews in the programming world.

In 2013, I heard Pete Goodliffe talk about becoming a better programmer, and he lined up panel of experts about how to become a better programmer. Having heard the talk, I endeavoured to put as much of it as I could in to practice. During one of the intervals, I had a chance meeting with Bjarne Stroustrup, who was gracious enough to agree to be interviewed for my blog. I was also encouraged to publish it in the ACCU magazine, so here we are.

If you're a C++ programmer, then Bjarne Stroustrup won't need any introduction at all. However, if you are new to C++, then Dr Stroustrup is the designer and the original implementer of C++. He is currently a Managing Director in the technology division of Morgan Stanley in New York, a Visiting Professor in Computer Science at Colombia University, and a Distinguished Research Professor in Computer Science at Texas A&M University. His best known published work is *The C++ Programming Language* which is currently in its 4th edition.

How did you get started in computer programming? Was it a sudden interest in computing? Or was it a gradual process?

> During my last year of high-school, I had to decide whether to go to university and if so what to study. I decided to study mathematics because I was pretty good at that in high school, but I wanted a practical form of mathematics, some kind of applied math. That way, I would be able to make a living doing math after graduation, rather than becoming a teacher. So, I enrolled in 'Mathematics with Computer Science' in my home-town university (Aarhus University) because I mistakenly thought that 'Computer Science' was a form of applied math. It was good that I was wrong about that because I wasn't as good at math as I thought at the time (though being a poor mathematician is better than not being a mathematician) and I absolutely loved Computer Science – and especially programming – when I eventually was introduced to it in my second year at university.

What was the first program you ever wrote? And what language did you write it in?

> In my first Computer Science course, we learned several languages and wrote tiny programs in those. I don't remember those exercises, though. The primary language taught was Algol60. The first program I do remember was a small graphics program written in Algo60. It was probably my first program that was not a set exercise. It connected points on a super-ellipse to draw pretty pictures. The 'user interface' allowed me to specify the parameters for the super-ellipse, the number of points, and the number of lines. From a programming point of view, it combined math with graphics.

What would you say is the best piece of advice you've ever been given as a programmer?

> Just one piece of advice? "Test early and often." But maybe that's just my own variant of the old Chicago advice about elections. ["Vote early and vote often: http://en.wikipedia.org/wiki/Vote_early_and_vote_often]
>
> Try not to be too clever: Bugs hide in complex code. Be clear and explicit about what you are trying to build, and how. By 'explicit', I mean 'write it down in good clear English that others can read'. Articulating a design is important and helps you when it comes to construct test cases and write assertions. Always think about how a piece of code should be used: good interfaces are the essence of good code. You can hide all kinds of clever and dirty code behind a good interface if you really need such code.

If you were to start your career again now, what would you do differently? Or if you could go back in time and meet yourself when you were starting out as a programmer, what would you tell yourself to focus on?

> I think I would have taken a year off to travel the world and improve my interpersonal skills. Had I known that I'd spend most of my career writing in English and giving talks in English, I might have paid more attention in my foreign language classes. On the other hand, I have found topics with no apparent practical use (such as literature, history, and even philosophy) at least as useful as many specific technical skills. It is good not to have too narrow a focus.

EMYR WILLIAMS

Emyr Williams is a C++ developer who is on a mission to become a better programmer. His blog can be found at www.becomingbetter.co.uk

Nothing is Set in Stone (continued)

However, we must embrace the fact that code changes: any code that stands still is a liability. No code is beyond modification. Treating a section of code as avoidably scary is counterproductive. ■

Questions

1. What particular attributes makes software easy to change? Do you naturally write software like this?
2. How can we balance 'no code ownership' with the fact that some people have more experience than others? How does this affect the allocation of tasks to programmers?
3. Every project has code that changes frequently, and code that changes little. The latter code may be staid because it's not used, because it is healthily designed for extension by external modules, or because people actively avoid the nastiness within. How much of each of these kinds of rigid code do you have?
4. Does your project tooling support your code changes? How can you improve it?

Becoming a Better Programmer: The Book

Pete's new book – *Becoming a Better Programmer* – is published by O'Reilly. It's available from http://shop.oreilly.com/product/0636920033929.do

> Bertrand Russell: "The time you enjoy wasting is not wasted time."

What was the biggest "ah ha" moment or surprise you've experienced when chasing down a bug?

I don't think this question applies. I dislike debugging and my usual reaction to finally finding a bug is "How could I be daft enough to write that!" Alternatively, "What was he/she thinking?" Often, a simple invariant would have caught the problem early or a slight restraint on 'cleverness' would have avoided the problem altogether. What I enjoy is to design and to express my designs in code. Sometimes, the realization of a design can be amazingly beautiful. 'Getting it' with the STL (Alex Stepanov's handiwork) was an "Aha!" moment. Discovering how to express the STL better with concepts comes close.

A lot is said about elegant code these days. What is the most elegant code you've seen? And how do you define what elegant code is?

I'd say that one of the best answers I've seen for what makes elegant code, is something I've read from ACCU's own Roger Orr:

```
}
```

Just that closing brace. Here is where all the 'magic' happens in C++. Variables get destroyed, memory gets released, locks get freed, files get closed, names from outside the closed scope regain their meaning, etc. This is where C++ most significantly differs from other languages. It is interesting to see how destructors – an invention (together with constructors) from the first week or so of C++ – have increased in importance over the years. So many of the modern and most effective C++ techniques critically depend on them

With the advent of C++ 14 upon us, where do you see C++ going in the future? Is there anything you'd like to see, or something you'd wish you'd done differently?

For the future, I'd like to see better concurrency support, concepts (requirements for template arguments), and increased simplicity. I'd like to explore the idea of simplicity within C++ with features such as range-for loops, auto, and libraries that make simple things simple. "Within C++, there is a much smaller and cleaner language struggling to get out" (and no, that language is not C, D, Java, C#, or whatever). I'd like to explore what such a "much smaller and cleaner language" might look like in general and how it could be embedded into C++.

The essence of C++ is that it provides a direct map to hardware and offers mechanisms for very general zero-overhead abstraction. A future C++ should be better at both. This precludes simple imitation of many modern ideas and trends. We can learn a lot from other languages (and always have done so), but direct import of language features is non-trivial.

The 'time machine question' is easier to answer because it has no effect on reality. We can't change the past and even if we could, I'm pretty sure I'm no smarter than 1980s-vintage Bjarne and he had a much better feel for what was possible at the time. Even the best time machine would not allow me to compile C++14 on a 1MB, 1MHz, 1985 computer. If I could have dropped the "Concepts Lite" design on Bjarne's desk in 1987, we might have avoided a lot of problems. At that time, I was looking at ways of constraining template arguments, so I would have recognized the importance of the ideas. Furthermore, the complexity and compile-time overheads are minimal so I could have implemented "Concepts Lite" well using 1980s technology. Of course, working from first principles, I would not have chosen the C declarator syntax or two-way conversions between fundamental types, but to fix those, you would need to take the time machine a few years further back for a chat with Dennis.

Try not to be too clever: Bugs hide in complex code. Be clear and explicit about what you are trying to build, and how

With technology moving so fast these days, where do you think the next big shift in computer programming is going to be?

Hard to say; there are so many different kinds of programming. I'm not even sure what the last big shift was. Dynamic languages? Object-Oriented Programming? Functional Programming? XML? Virtualization? C? Generic Programming? I'm pretty sure I could point to areas where each of those answers would be quite reasonable – as well as areas where each would be ludicrous. The field of software development is just too huge and diverse for simple generalizations.

In the parts of the C++ world that I know best, my guess is that the improved support for concurrency in C++17 (various higher-level models of concurrency beyond the basic threads-and-locks level) will cause major changes and that concepts (starting with 'Concepts Lite') will complete generic programming's move into the mainstream. That combination, coming on top of the improvements provided in C++11, should completely change the way C++ is used.

I say "should" rather than "will" because I fear that many will hold back out of fear of novelty, for lack of intellectual flexibility, or because of constraints from old code bases. People are more likely to see the risks and complications of a change than to appreciate the risks and costs incurred by using outdated tools and techniques. Maybe, I'm being a bit less optimistic than I should be: people who have used C++11 tend to complain bitterly when they have to go back to C++98. Significant progress has been made and we can now write simpler and better code than we used to. Many people know that and they are not going to accept "the old ways" forever.

Finally, do you have any advice for any kids or adults who are looking to start out as a programmer?

Don't just program. Know what problems you want to solve using programming. Don't rush into programming. Work on your communication skills. And don't forget to have fun – you are much better at things you enjoy doing than things you consider tedious. Learn to see the beauty in elegant and efficient code! ■

A Tour of C++, for people who want to quickly know what C++11 is, and his textbook for beginners: *Programming: Principles and Practice using C++*, which now uses C++11 and some bits of C++14 is available now from your usual book reseller.