An interview with Bjarne Stroustrup by the CCF (China Computer Federation) in November 2014.

- What were your major original design goal of C++? And what's the most memorable part in the design of C++?

  ➔The major original (as well as current) design goals were to allow me to write maximally efficient systems' code and control complexity, to write elegant and efficient code, to provide direct access to hardware and zero-overhead abstraction. That's three ways of saying essentially the same thing that I have used over the years.

  If I have to point to one key feature of C++, it must be constructor and destructor pairs. Those are the basis for most characteristic C++ techniques – modern and from the earliest days. Those are the basis of C++'s support for implicit, predictable, and general resource management. Those underlies C++'s use of constructors to establish class invariants. Those are the basis of C++'s exception safety (through RAII). Those are the key mechanism in C++'s "smart pointers" that provides an alternative to garbage collection.

  Curiously enough, designing this wasn't hard and constructors and destructors entered C++ in its first month as an obvious feature.

- Over the years you have put quite a lot of effort on the ANSI/ISO standardization of the C++ programming language. Can you please give a brief overview of your focus and how the standardization process works? When did you find yourself reaching deepest into your emotional reservoir to overcome a difficult challenge to your vision?

  ➔Standardization is a hard, but important task. For C++, the standards committee is the focus of efforts to make C++ a better tool for serious programmers. The committee meets three times a year – for a week each time – and much email and many documents are exchanged in-between meetings to ensure progress. About 100 people come to each meeting and about 200 more takes part in the electronic exchanges. We aim for consensus: We are not happy with less than a 90% support for a proposal. Proposals that do not meet that exacting standard are usually considered in need of more work. The result of this effort has been that all implementers stick to the standard even though there is no legal framework to force people to follow an ISO standard. Members of the committee mostly work in industry and represent organizations from many countries, many industries, and of widely varying sizes. We have people representing huge corporations, such as Google, IBM, and Microsoft, but also individuals representing themselves. It is an all-volunteer effort.

  Choosing among the many directions that the language and its standard library should evolve is very tough for a group of people with diverse backgrounds. Some people wants every feature from every fashionable programming language and some wants hardly anything new. I try to take a lead in that wherever the technical area is one I master (For examples and details, see my papers from the ACM History Of Programming Languages (HOPL) conferences – there are links on my publications page). Building up a consensus is a very

hard part of the standards effort. Individual proposals can be technically hard to craft – especially because we work under the constraints of performance, usability by "ordinary programmers," and backwards compatibility. C++ is not an academic experiment in abstract beauty, but a real-world tool. Making progress takes not only insight, but also patience.

Sometimes, I have had to wait for more than a decade before I could get an important improvement sufficiently refined to be accepted. For example, I first looked at the problem of specifying the interface of a generic function in 1988 or so. I could not come up with a solution that met C++'s needs of generality, zero-overhead, and precisely specified interfaces. Starting in 2002, I – together with others and inspired by the work of Alex Stepanov – tried again. That effort failed because the design became excessively complex and too hard to use. It was rather painful to scrap the work of many people over many years as an experiment that failed. Finally, in 2011 we got onto a better track based on the idea of specifying requirements for template arguments as compile-time predicates. Dozens of people were involved in discovering the basic principles and use patterns. Together with Andrew Sutton and Gabriel Dos Reis, I designed the language facilities for the new approach. The new approach was often referred to as "Concepts Lite" to distinguish it from the old approach and to emphasize its smaller size and ease of use. Andrew Sutton implemented "Concepts Lite" in a branch of GCC and I expect it to become an ISO Technical Specification ("a TS") any day now.

- What do you thinking of formal methods? How's formal method used in the design and implementation of C++ programming language?

  ➔I – together with friends – have looked at formal methods repeatedly over the years and used it for specific tasks, such as specifying to memory model, constant expression evaluation, and the specification of template argument requirements ("concepts"). However, we have not found such models applicable for the language as a whole. The key reason is that C++ (as I think of it) is based on abstractions based on well-specified interfaces, such as the real machine, rather than on building up facilities from a mathematical abstraction.

  I suggest you look at the work of my former colleague and current collaborator Dr. Gabriel Dos Reis, especially his POPL papers, for an idea of how formal specification can be used for C++.

- Can you please give an quick overview of C++ 11 and C++ 14: what are the key language features and why were they added – the fundamental concept you want to address?

  ➔Now that's a huge topic! C++11 was a major new release of C++ that made C++ feel like a new language. C++14 was a relatively minor improvement of C++11, completing the work of C++ based on a few more years of experience and some C++11 use.

  Before going into details, I'd like to point out that C++11 and C++14 are not science fiction: complete implementations are shipping (e.g., GCC and Clang) and others are close to complete (e.g., Microsoft). This is remarkable because it is still 2014 – the implementations are slightly ahead of the formal ISO approval process.

Also, there is a lot of information about C++11 and C++14 “out there.” A quick web search will find many lists and articles. I recommend my C++11FAQ and especially the ioscpp.org C++ FAQ are reliable sources, but they are not alone! For people who want an overview of modern C++, I recommend my *A Tour of C++*. It gives an overview of all of C++ and its standard library in just 180 pages. People aiming for mastery should look at my *The C++ Programming Language (Fourth Edition)*. Novices and programmers with a weak C++ background should consider *Programming: Principles and Practice using C++ (Second Edition)*. All three books should be available in Chinese this year or the next.

When looking at lists of C++11/14 features, please remember that a feature is not meant to be used on its own. They are designed to work together for a simpler and more elegant expression of solutions to problems.

- ⇨ Threads
- ⇨ Auto (deducing an object’s type from its initializer)
- ⇨ Range-for (a simple notation for the simplest loops)
- ⇨ Initializers and constructors
- ⇨ Constexpr (Generalized compile-time computation)
- ⇨ Move semantics
- ⇨ Lambdas
- ⇨ Template aliases
- ⇨ User-defined literals
- ⇨ “minor improvements”: unions

Standard library enhancements

- ⇨ Regular expressions
- ⇨ Hash tables
- ⇨ Timing support
- ⇨ Random numbers
- ⇨ file system support
- ⇨ “smart pointers” (shared_ptr, unique_ptr)
- ⇨ static reflexion (still incomplete)
- ⇨ move semantics, emplace, initializer lists

It is hard for me to succinctly describe how each of these many features fits into the overall language. Several features serve to make initialization simpler and more flexible: auto, initializer lists, variadic templates (when used for constructors), even move semantics because often the result of a function is used as an initializer or as a function argument. Lambdas and template aliases simplifies the specification and use of algorithms. The generalization of constant expression evaluation (especially constexpr functions) provides a replacement for complicated template metaprogramming for compile-time evaluation leading to values.

Using these features – in combination with older features – allows for simpler, more typesafe, better structured, and sometimes faster code.

Consider this function:

```
template<typename C, typename V>
vector<Value_type<C>*> find_all(C& c, V v)  // find all occurrences of v in c
{
        vector<Value_type<C>*> res;
        for (auto& x : c)
                if (x==v)
                        res.push_back(&x);
        return res;
}
```

The function **find_all()** fills the vector **res** with pointers to every element of the container **C** that equals **v**. In C++98, returning a vector was a copy – a potentially very expensive operation. In C++11, this return has been optimized to a simple, efficient move operation. The return is achieved using vector's move constructor that avoids copying and also avoids explicit memory management. The user doesn't have to see pointers and worry about **new** and **delete**. We can test **find_all()** like this:

```
string m {"Mary had a little lamb"};
for (const auto p : find_all(m,'a'))        // p is a char*
        if (*p!='a')
                cerr << "string bug!\n";
```

Now look at a version of the classical "draw all shapes in a container" example. Since Simula67, this has been used as an illustration of object-oriented programming. I make a few shapes and draw them:

```
void draw_all(Drawable_sequence& s)
        // the classical "draw-all Shapes" example
{
        for (auto x : s) s->draw();
}

void test()
{
        vector<unique_ptr<Shape>> lst {         // Shape is an interface
                                                // unique_ptr represents ownership
                make_unique<Circle>(Point{0,0},42),
                make_unique<Triangle>(Point{20,200}, p2,p3),
                make_unique<Square>(Point{40,40},42)
        };

        draw_all(lst);
}
```

Note the use of **unique_ptr**s to represent ownership and guarantee proper deletion of the shapes at the end of **test()** and the use of a range-for to traverse the list. **Drawable_sequece** is a concept I defined specifically for representing containers of things that can be drawn, such as a **vector** of **unique_ptr**s.

- Our hardware has stepped into the multi-core or many-core era, but the parallelism support is still far from satisfying. How do you think existing programming language should be enhanced? Or, could you provide some principles for next generation programming languages for language designer?

  ➔I think we need to support several models of concurrency and parallelism. There is not one model that fits all uses. For example, the concurrency needed for a GUI differs from that needed to support vector parallelism, which differs from that needed to support concurrency within an operating system kernel. Today, we often do it all by dropping to the lowest common level: threads and locking, which serves all areas badly. We can do better. I think that to support many high-level models of concurrency, we can't build them into the language themselves. Instead, we need to build good, relatively low-level mechanisms into the language and then build higher-level libraries on top. C++11 gave us type-safe threads and locks support. It also gave us primitives for portable lock-free programming. I expect C++17 will give us several higher-level models of concurrency and parallelism.

  Concurrency (often in the form of many cores) and parallelism (often in the form of enhanced vector units as part of the cores) is the main areas of performance improvements in modern chips, so to benefit, we must match those hardware improvements with advances in software.

- Do you think it possible to involve some features in existing languages to support crowd sourcing?

  ➔Crowd sourcing works best for well-understood problems and established tools and procedures. I have less hope for crowd sourcing for language design. "The crowd" has infinite appetite for novelty, but little taste. That said, we can benefit hugely from more people helping out with the implementation and testing of libraries. This is already happening.

- How would you evaluate a programming language, a.k.a., what make a beautiful programming language?

  ➔For many years, I used to say "Even I could design a language that is much more beautiful than C++." My aim was always more for a useful language than a beautiful one. Obviously, I prefer beauty where I can get it, but not at the cost of usability (e.g., performance, compatibility, and generality).

  ???

- There are many programming languages developed these days with different focus and supporting a different set of features. From your perspective, what do you think are the most important principles we need to bear in mind when developing a new programming language?

  ➔Decide which problem or problems the language is going to help solve and make sure you do that well. Beyond that, I favor type and resource safety, expressiveness, performance, and interoperability with other languages and systems.

  ???

- Object oriented programming language is still the dominate language used in the information technology industry. Recently we saw the industrial adoption of functional programming languages and dynamic languages (like Scala, Ruby). What's your view of the direction of programming languages in the next decades for industrial software?

  ➔I'm not a fan of idea that there is one language or even one style of language that is good for everyone and everything. The solution to most challenging real-world problems are best solved by a combination of programming languages and by a combination of styles within languages.

  For C++, I will adopt effective techniques from a variety of programming languages (with acknowledgements, of course) and try to fuse them into a synthesis that will help C++ programmers. Most programs I write combines aspects of object-oriented programming, generic programming, functional techniques, and procedural programming.

  I hope the industry and academia will wake up to the fact that different tasks require not just different languages, but that they require different skills and different education. The notion that "programming paradigms" come and then gets replaced is wrongheaded and very costly. I remember Kristen Nygaard (the inventor of object-oriented programming) to try to explain that, but most people were too excited about the new (OOP) to listen.

- What do you see as the most promising directions for future work in programming language? What interesting and important open problems do you see?

  ➔We have to address the needs of programmers using concurrency in all of its forms. In particular, we have to use programming models that offer greater simplicity and greater performance than plain old threads and locks. The ISO standards committee has several "study groups" working on that. In C++11 we got support for lock-free programming and type-safe threads and locks. For C++17, we'll get more, but it's too soon to say exactly what. We are working on libraries for work-stealing, for co-routines, for SIMD vectors, for futures with continuations, for transactional memory, and more. Something major is going to come out of that for C++17. The progress in machine architectures emphasized SIMD and many cores, so we must address those.

Soon, we'll have "concepts", that is, direct language support for requirements on template arguments. That will change the way we think about generic programming. Concepts are already available as a branch of GCC and work is in progress for Clang and Microsoft implementations.

From a language point of view, we need to enhance type safety and expressive power without damaging C++'s ability to deal directly with hardware. Obviously, performance should also be improved.

Dramatically improving compile times is another challenge. A standards committee study group is looking into that.

- We have put a lot of effort to research trustworthy software, from different views like formal methods, programming language, runtime infrastructures, and software engineering methodology. From programming language perspective, what's your view how we can make our software more trustworthy?

  ➔First we must get over the notion that programming is a low-level skill, preferably done by low-cost labor. Without professionalism the promises of various technologies will be compromised by developers who value expediency and low-cost over quality.

  We need to work towards complete type and resource safety. We need to use a more type-rich style of code: using an integer as an integer is type safe, but not expressive enough to allow the compiler to catch silly mistakes because everything can be represented as a set of integers. We need to get a better handle on concurrency. Static analysis is a most promising area, especially when combined with code transformation. I am not a fan of languages that seriously restrict what can be expressed (in the name of safety) or languages that impose non-trivial run-time costs: they will be ignored in favor of faster and unsafe languages and techniques. I'm pushing C++ in the direction of a better and more expressive type system – without run-time overheads.

- What's your suggestion to the PhD candidates who are doing research on programming language, formal methods, and software engineering?

  ➔I'll repeat the advice I got from Maurice Wilkes and Roger Needham (look them up) when I suggested that I might like to study those areas: "Don't. The success rate in those areas is too low." Try solving some more concrete problems (e.g., networking, real-world security, graphics, etc.) and then apply your knowledge on languages and tools if you need to. The average PhD student does not have sufficient experience to distinguish between real and imaginary progress in programming techniques and tools. Obviously, there are exceptions – and most students and most professors think they are the exception – but the odds are low.

  We badly need progress in these areas, but I don't think we'll get it from another 1,000 PhD students right out of undergraduate courses. I think we need industrial experience and expertise.

- finally, do you have anything else you want to share with the young scientist and researchers in China Computer Science Community?

  ➔Try to make a difference in the world! Don't just look at grades and starting salaries. After all of these years and all of this time spent in industry and academia, I'm still an idealist. Call me naïve, if you must, but someone has to be. Significant changes take time – often decades. Find a worthwhile "grand aim" and work on it over the years. Most likely the needs of careers and family will get in the way at times, but never completely lose track of your ideals.

  Our civilization depends on computers, hardware and software, and it has better be reliable, dependable, and affordable software, or we will all be in deep trouble. Do something to help.

  And don't just do computing. Have other interests that can motivate and inform your work. What is it that you'd like computers to do better? What will make the world a better place? Maybe you should study history, biology, sociology, mathematics, music, whatever. Don't just get lost in technical details of computing. Every program is meant to benefit people somehow.