Orriginal English version of interview of Bjarne Stroustrup by Andrea Leganza for Il Programmo Magazine; Sat, 24 Oct 2009 20:21:24 +0200

1) Thank you for your willingness, let's start: what are you doing now, projects and work.

> Working to finish C++0x (the next ISO C++ standard), doing some academic research, and teaching programming to freshmen using my new book *Programming – Principles and Practice using C++*. The academic research mostly relates to static analysis and transformation of C++ programs (see my publications list on my home pages) and the work on C++0x includes producing tutorial material, such as my C++0x FAQ and (research and popular) articles based on my exploration of the new features.

2) In the 2009 you celebrate, and the community too, the 30th anniversary of the birth of the language (on 1979 you started to work on it), what's changed on the IT in these 30th years that you never imagined?

> That’s a long time! A lot has happened since 1979 (some of it significantly helped along by C++) and fortunately C++ has also evolved to keep up with the changes in its core application areas.
>
> The design of C++ anticipated the PC revolution and the increase in memory sizes and processor speeds (at least some of that; the effects of an exponential explosion are hard to imagine even if you can see it coming. I anticipated the increase in size and complexity of applications, but not the significant increase in Web applications. I always focused on what you might call “infrastructure.” Systems programming and embedded applications belong to that category. When I started, telephone switches, compilers, and operating systems were iconic examples. These days browsers, virtual machines, and games engines have more intuitive appeal, though the “old” application areas still underpin all. I’m very pleased to see C++ holding its own in its intended application space, partly by evolving to meet new challenges.

3) You wrote, as an answer in your FAQ about the origins of C++, that Simula is the true father of C++, and Smalltalk a sibling, there is some language now available that you believe is the true son of C++?

> Essentially all modern languages – at least all that claim to support some notion of “Object Oriented Programming” – owe a great debt to Simula. Too many people forget that and talk as if all the good ideas in their favorite language sprang fully formed from the fertile brains of its designers. The other major base for C++ was of course C. As a field, computing unfortunately has a poor sense of history. I think that’s a sign of immaturity and impedes communication among practitioners. Fields like Math, Medicine, and Physics do better: You couldn’t claim to be a physicist without knowing what Newton did and why it matters or a biologist without appreciating Darwin. I feel that all programmers, not just C++ programmers, ought to know the names “Ole-Johan Dahl,” “Kristen Nygaard,” and “Dennis Ritchie” – and have at least a minimal understanding of what they did for modern software.
>
> New languages these days tend to be object-oriented in the sense that they have inheritance and reference semantics or alternatively come from the functional programming tradition. On the other hand, C++ relies on true local variables, scoped resource management based on constructor/destructor pairs, and a lot of overloading to get generic programming working smoothly. Also, and essential for many C++ applications (but not inherently in its programming model), C++ has a direct mapping to hardware.

There are many, many languages borrowing from C++ (often without acknowledgement), but few that are close enough to be considered direct descendants. Maybe I'm thinking too narrowly and might one day wake up to realize that something was "a true son of C++" in some sense that I have missed. I hope so. That happened to Kristen Nygaard (The father of object-oriented design and programming). At a conference, he suddenly jumped up and gave a small speech "I have wondered about this guy, Bjarne, who claimed to carry the flame of Simula … now I see it! …" (after quite a few years☺). He then hauled me up and gave me a bear hug – note that Kristen was 6'4 (or more) and felt even bigger. He was a giant in every way.

4) C++ is a multi-paradigm programming language that supports object oriented programming; there is some aspect/feature that you believe is underestimated about your language by the majority of the developers community?

I'm getting tired of the "multi-paradigm" label. For me, it was never more than a stopgap until we could gain a better understanding of how to design and program with the set of tools provided by C++. Look at it this way: C++ has slowly been growing to support a wide range of programming techniques and a lot of that effort has focused on how to refine language features so as to enable a smooth integration of the various techniques in a simple program. When I program, I invariably end up using a combination of free-standing small classes, some classical object-oriented hierarchies, and parameterized classes and parameterized algorithms as you find in generic programming. I do not choose among "paradigms" and use the most appropriate for a whole program. Rather, I select my programming tools on a much finer-grained basis so as to best express solutions to my design problems. That just gives better programs (better structure, easier to understand, better performance, simpler maintenance, etc.). The way I see it is that I map ideas to code using the appropriate combination of traditional techniques for each idea. There is nothing "multi" about that, but I have yet to properly characterize my approach.

In my view, good C++ programming is about finding/designing appropriate abstractions for the entities in our programs. Its strength is in defining flexible and efficient ("lightweight") abstractions. Most of the time, you don't need to build massive class hierarchies or fancy template meta-programming tools to benefit. A simple Point class or resource handle can make all the difference to the simplicity of code and such classes can be built and used with zero overhead compared to hand-written C code. So I see C++ as a language for design and programming based on lightweight abstractions. The lack of overhead is most relevant in resource constrained applications, so I see C++ s a language for software infrastructure. Those application areas are also the domain of professionals so even though I care for novices (of all backgrounds) simplicity of language features is not my most important aim; the simplicity of applications is ultimately more important.

What is underestimated? The use of constructor/destructor pairs to support scoped resource management deserve far wider notice than it gets. It is the basis for most modern C++ programming techniques. Containers save us from the complexities of "naked" new and delete operations. Smart pointers save us from undisciplined pointer use. The unfortunately named RAII ("Resource Acquisition Is Initialization") idiom, simplifies the management of higher-level resources, such as locks, sockets, file handles, and transactions. Programmers who miss that are doomed to write far more complicated code and typically far more code, just to produce more bugs and less maintainable code.

5) Did you miss some aspect/feature you didn't put on this language?

There are many “things” that I would like, but many can be provided as libraries (e.g., an infinite precision type) and others are “system properties” rather than individual language features (e.g., perfect type safety). If I have to pick just one language feature, I’ll point to multi-methods, that is, the ability to dispatch on the (dynamic) type of more than one argument type. For example, **intersect(s1,s2**) chooses the appropriate “intersect” for **s1** and **s2**, so if **s1** is a **Circle** and **s2** is a **Triangle**, we invoke a function such as **intersect(Circle,Triangle)** if such function exists. I have been thinking about this for a long time (decades, see D&E), but now, together with my students, I have a design with an experimental implementation which has been experimentally validated: Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup: *Open Multi-Methods for C++*. Proc. ACM 6th International Conference on Generative Programming and Component Engineering (GPCE). October 2007. And soon: Peter Pirkelbauer, Yuriy Solodkyy, Bjarne Stroustrup: *Design and Evaluation of C++ Open Multi-Methods*. Science of Computer Programming. Elsevier Journal. To appear 2009. One way to explain potential the importance of this is that if you have multi-methods, you’ll never again have to implement a visitor (a use of the visitor pattern). And in the best C++ tradition, a multi-method call with two arguments is faster than two virtual function calls and the generated code is smaller than what you get from workarounds.

Of course many of the features I miss most in C++ are part of C++0x and are already appearing in implementations, such as GCC and Microsoft. Examples are concurrency support, uniform and general initialization, general constant expressions, efficient smart pointers, regular expressions, etc. (See my *What is C++0x* paper and my C++0x FAQ).

5.5) What do C++0x offer to new C++ programmers? To expert C++ users?

Those are big questions! My C++0x FAQ and my recent article *What is C++0x?* provide some answers, but here are a few examples. Consider writing a simple loop to output the elements of a list. In C++98, we would write something like:

```
for(std::list<Elem>::iterator p = lst.begin(); p!=lst.end(); ++p) cout << *p << "\n";
```

We could write the same in C++0x, of course, but we could also simplify:

```
for(auto p = lst.begin(); p!=lst.end(); ++p) cout << *p << "\n";
```

In C++0x, we use **auto** to indicate that the type of a variable should be that of its initializer. The two examples above are equivalent. However, in C++0x, we don’t have to remember or type **std::list<Elem>::iterator**. This simplification is the oldest C++0x feature. I designed and implemented it in the winter of 1983/84, but was forced to remove it because of C incompatibilities. We can simplify further. In the simplest and most common case where we just want to visit each element of a sequence, we can use the new range-for loop:

```
for(auto x : lst) cout << x << "\n";
```

This means “for each element **x** in the sequence **lst** do …”. This works for any sequence; that is, for anything for which we have defined a **begin()** and an **end()**. The use of **auto** ensures that the type of **x** is the type of the elements of the sequence.

Another simplification is the provision of a general and uniform initialization syntax and semantics based on **{}** initializer lists. Such list can be used everywhere in C++ where an object can be created and a created object. **X{v}** gets the same value wherever that notation is used. For example:

```
vector<string> designers { "Dahl", "Nygaard", "Ritchie" };
auto p = new vector<string>{ "Dahl", "Nygaard", "Ritchie" };

struct S {
        vector<string> vs;
        int n;
};
S s { { "Dahl", "Nygaard", "Ritchie" }, 3 };    // as struct initializer

class Strange : S {
public:
        S(vector<string> s) : S(s,s.size()) { }    // as base initializer
        // …
};
S ss { "Dahl", "Nygaard", "Ritchie" };

void f(vector<string> s);
f({ "Dahl", "Nygaard", "Ritchie" });  // as argument

vector<string> g()
{
        // …
        return { "Dahl", "Nygaard", "Ritchie" };      // as return value
}
```

Of course, this is nonsense code. It simply illustrates the point that the **{}** initialization can be used everywhere. If can also be used for all types; for example:

```
string a[] = { "Dahl", "Nygaard", "Ritchie" };          // an array
string name { "The earth is not flat" };
int x{42};
complex<double> z { 5,4 };
```

For simplicity, I used literals as list elements, but any value will do:

```
vector<string> vec(const char& a, const string& b, const string& c)
{
        return { a, b, c };
}
```

These simplifications (**auto**, range-**for**, and uniform initialization) help both novices and experts, but you might say “that’s just improved notation, show me something I couldn’t do in C++98!” I think the “just notation” point underestimates the value of good notation – after all, if we had sufficient time and patience we could do everything in assembler. However consider the problem of getting a large new object out of a function, say getting the result of a matrix addition returned. For performance reasons, we conventionally allocate using **new** and return a pointer, pass in an object in which to place the result, or resort to clever special-purpose trickery. Each solution comes with a set of problems and pitfalls. Often some overhead is involved. What we need is a simple, general-purpose solution, so that we can write:

```
Matrix operator+(const Matrix&, const Matrix&);
Matrix m = mx+my;    // mx and my are matrices
```

No matrix copies should be performed. C++0x offers a simple way to do this. The implementation of the **Matrix** add can be something like this:

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
        Matrix res;
        // res[i] = a[i]+b[i]
        return res;
}
```

This looks as if we are copying **res** when we return it, but we are not. The idea is to give **Matrix** "move semantics." That is, if we are about to copy a **Matrix** and then immediately afterwards destroy the original, we move it instead. That's achieved by giving **Matrix** a move constructor in addition to the conventional copy constructor:

```
class Matrix {
        double* elem;  // the elements of the matrix
        int sz;        // the total number of elements
        // …
        Matrix(const Matrix& a);        // copy constructor
        Matrix(Matrix&& a);             // move constructor
};
```

The **&&** indicates an "rvalue reference; "that is, a reference to something that's about to be destroyed – some object we don't need the value for after moving it. Compare the obvious implementations of the copy and move constructors:

```
Matrix::Matrix(const Matrix& a)         // copy constructor
        :elem(new double(a.sz), sz(a.sz)        // get new space
{
        for(int i=0; i<sz; +=i) elem[i] = a.elem[i];        // copy elements
}

Matrix::Matrix(Matrix&& a)              // move constructor
        :elem(a.elem), sz(a.sz)  // use old space
{
        a.elem = 0;        // no copies, just make the matrix empty
        a.sz = 0;
}
```

For a large **Matrix**, say with 100,000 elements, the performance difference is huge. We can use this technique for every class that essentially is a handle to data on the free store. The standard library containers, such as **vector**, **list**, and **map**, now have move constructors, so you can start benefitting without writing a single rvalue reference of move constructor. In fact, you might find that existing code speeds up as older code implicitly benefit.

You might consider matrices esoteric, but unless you have been vacationing somewhere without news sources for a decade, you'll have heard that we need to cope with concurrency as provided by multicore processors. We have been writing concurrent programs in C++ for a couple of decades now, but until C++0x the standard didn't offer any guarantees about it or any standard features for expressing concurrency. C++0x offers a precise statement of what C++ does on concurrent hardware, atomic types for low-level concurrent programming, a **thread** class, **mutex**es, condition variables, and **lock**s for systems level programming, and **async()**, **future**, and **promise** for simple higher-level concurrent programming. You can look up the details in my C++0x FAQ or in the draft standard itself. Here, I'll just show a simple example of the use of the **async()** "launcher" of concurrent tasks and the **future** type for simple return of results:

```
double accum(double* b, double* e, double init);   // compute the sum of [b,e) and init

double comp(vector<double>& v)          // spawn many tasks if v is large enough
{
        if (v.size()<10000) return accum(&v[0], &v[0]+v.size(), 0.0);

        // spawn tasks to do each of the four quarters of v:
        auto f0 = async(accum, &v[0], &v[v.size()/4], 0.0);
        auto f1 = async(accum, &v[v.size()/4], &v[v.size()/2], 0.0);
        auto f2 = async(accum, &v[v.size()/2], &v[v.size()*3/4], 0.0);
        auto f3 = async(accum, &v[v.size()*3/4], &v[0]+v.size(), 0.0);

        return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Here, **async()** is used to "spawn" concurrent tasks and **get()** waits for results (if waiting is needed). We could write this "by hand" using threads and locks explicitly, but for simple tasks, why bother? Also, **async()** could easily be smarter about the use of threads than the programmer because its implementation may have access to more information (e.g. how many threads are already in use and how many processor cores are already busy). If you want more detailed control, just take it.

6) When C++ started to be a programming language widely used around the world there weren't so many languages available, now every year we see some new one that claims to be better/faster/stronger that all the previous, this increasing number of programming languages is something to appreciate or not?

The average number of new programming languages each year over the last 50 years has been around 200. You can adjust that number up or down by choosing different definitions of "programming language" (What's a language and what's a dialect? How widely used does a language have to be to be counted? Etc.), but the total number is huge.

I strongly appreciate the proliferation of experimental languages and of languages that address problem areas poorly supported by older languages. I have far less sympathy for languages developed with minimal innovation to lock users into corporate platforms. Such languages typically serve their users well (in the short term), but does harm to the computing profession at large by making programs and skills less transferable (applicable only on a single platform), by impeding communication of ideas, and confounding education. The multiplicity of similar languages is of course also expensive to produce and maintain, but corporations seem to prefer to spend millions on a proprietary languages over spending thousands on a language usable across platforms.

7) There is a language that Bjarne would have wanted to create (obviously with the exception of C++)? And why?

> I would like to see a completely type safe and much smaller language following the C++ model. That is, a language with a low-level machine model, zero-overhead abstraction mechanisms, true local variables of all types, and constructor/destructor-based lifetime control. I think it can be done, but it would be distinctly non-trivial. The final section of my paper for the ACM History of Programming Languages conference (HOPL-3) goes in slightly greater detail.
>
> Let’s not forget that today’s C++ is a far more powerful tool than my original release 1.0. C++0x (the upcoming revision of the ISO C++ standard) is an even better tool for real-world applications.

8) Java is become at the start of the millennium synonymous of Object Oriented Language and is used now in many Italian universities as the only way to teach object oriented programming, while I started my voyage in the programming world with C++, and after that followed C and many years later Java: I feel they are losing something learning only Java, using this language so intensive is pushing them to rely too much on garbage collectors and automatic operations/optimizations, they don't care on performance and memory footprint of their applications: simply put...they think less on this superficiality is seen on many applications, what do you think? Do you believe that this language have decreased the quality of software?

> I don’t know if it is correct to blame a language for dumbing down curricula. For a variety of reasons, many people wanted simpler curricula for computer science, programming, IT, etc. and Java was there (heavily marketed by Sun) to use for good and bad. Some wanted cheaper (less educated) programmers; some wanted a “gentler” computing curriculum to attract a larger group of students; some thought that a traditional focus on systems issues, algorithms, etc. was outdated. Sun rode those waves and fueled them to promote Java.
>
> Whatever the reasons, the effect is a generation of students that does not feel comfortable with systems issues, is often afraid of using languages that run directly on a machine, is unfamiliar with general resource management issues, and often feels that such issues are irrelevant or even demeaning (“someone or something else should do that!”). It is not inherently a bad idea to train some (but not all) students to be purveyors of fairly standard solutions to fairly standard problems, rather than as professionals capable of handling most problems based on fundamental concepts and general techniques. In fact, the massive Java infrastructure (and similarly .Net) and training lots of people to use it (them) have probably have led to higher quality software in many cases. However, letting people with such backgrounds lose on traditional systems programming tasks and equivalent embedded systems task has often been less than successful. That’s a great pity – and a potential disaster – because a very substantial amount of current and future work is in those areas. In particular, every “advanced applications and environments” is enabled by an infrastructure which is typically written in C or C++. My opinion is that, given suitably educated developers, infrastructure problems are typically best handled with C++.
>
> So, in addition to the “standard application development” tools and developers, there should be an education for the professionals building infrastructure and advanced applications. Those

professionals should be supported by high-quality tools. That's what C++ is for and is best at, but unless we teach modern C++ programming techniques these areas will be built with serious problems and high costs. For example, letting a Java programmer lose with C++ without some serious education or guidance, typically results in a mess of pointers and resource leaks even beyond what we used to se with C.

Lately, motivated by my work in academia, I have written a bit about education of software developers: Bjarne Stroustrup: *What should we teach software developers? Why?* To appear in the January 2010 issue of the CACM. Bjarne Stroustrup: *Programming in an undergraduate CS curriculum*. WCCCE'09. May 2009.

9) in these years if I ask someone what language he will use to develop a program for desktop PCs quite all will answer Java, someone .Net, if i ask them if they will have to develop for embedded many will answer again Java, but when i reply about the resources problems on using this technology on embedded they don't know what language answer, or ,with fear, they answer "C or C++?": the consciousness of today programmers of what to use on different platforms is really so decreased due to the spread of this Virtual Machines and so high level languages?

Yes. That fear of the machine will lead to worse software and/or the work migrating to places where there are sufficient concentrations of programmers without such fear. A wider understanding of and use of modern C++ could counter those trends. Please note that C++ in itself that cannot help – C++ written in Java or C style would simply exacerbate the problems. What would help it is the use of the modern design and programming techniques that C++ directly supports. Too many people have a 1985 vintage view of C++ and unfortunately often perpetuate such views.

10) The "iPhone trend" programming is increasing Objective-C popularity, what do you think about this "language" (our programmers knows that isn't a real language but an object oriented extension to C)?

Objective C is primarily an Apple mechanism for locking in users and ensuring that code and skills are not portable to other systems.

11) If you could throw one down from a cliff, which one would be? .Net, Java or some else? And why?

I just don't do that.

12) What should be the correct programming path to follow to become a really good programmer?

There is no just one right way. That's obvious, I hope. I think the key is a love of programming, which must be channeled into a love for clear logical thinking, and the love of some application area.

Know at least two rather different language (and preferably more) well. Have a solid understanding of an application area ("solid" as in having an undergraduate degree or "minor" in it).

Read a lot of code and a lot of technical literature.

Don't be a language bigot.

Learn from a good designer/programmer, read a lot of code, and write a lot of code. However smart you are, your first code is going to be pretty awful.

Persist.

13) When I start to learn some programming language I read about three four books about it, at least, now many programmers learn using web resources from "experts" that many times do "copy and paste" from other resources, I believe a book is the best resource to start, and the web resources should be the second step in the learning process, do you agree?

I agree. What you need to master is new techniques and idioms. Focusing on syntax and trying to do things in familiar ways is a trap. When choosing from simple “how to” sources, we are likely to select what we already know and fail to understand the new. I see no substitute for technical books and articles for learning to appreciate new concepts and techniques. A professional doesn’t just know how to do things; he/she must know why something is best done one way rather than another. The online sources are invaluable once you get to try out the concepts and techniques in practice.

Many “experts” on the web are at best unreliable and typically seriously understate the problems of real-world software to be able to offer simple solutions.

However, reading three books may be excessive. It takes only one good book to get started with a language and reading too much too soon could distract from the important task of writing some simple code to try out ideas. Programming requires a combination of theoretical and practical knowledge.

14) With the exception of the books you've written, starting from The C++ Programming Language, which one of yours you suggest to be the second one to buy?

Of mine, I’ll recommend *Programming – Principles and Practice using C++* for people unacquainted with C++ and relatively inexperienced as programmers. Recommending from the huge mass of good books written by others is hard because I don’t know the potential reader’s background, interests and needs. If you are interested in algorithms, have a look at Stepanov and McJones: *Elements of programming*. If you are interested in graphics, maybe have a look at a QT book (there are several and I don’t have a preference) or an Open GL book (there are several and I don’t have a preference). For multi-threading, multi-code programming, I’m looking forward to Anthony Williams’ *C++ Concurrency in Action: Practical Multithreading* (scheduled for early 2010 and currently available “in beta” on the web). There really are so many different needs and interests that it’s hard to recommend. My “C++ In Depth” series has several good books, such as Sutter and Alexandrescu’s *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*.

Always work hard at understanding your application area. This could lead to reading up on just about any topic, from accountancy, to biology, to space flight. If you are interested in anything “close to the hardware,” be sure to be up-to-data with machine architecture (because our assumptions about performance are so often wrong or outdated). For example, read one of Hennessy and Patterson’s recent books. Most good technical books are not language specific.

15) Many projects are build in C++, like the really nice QT (which is a cross-platform application and a UI framework), now owned by Nokia, there is a favorite one? And why?

There are so many applications to choose from! For example, see my applications page. I guess my favorite applications are those you can see, such as the Mars Rovers, the ships engines, PhotoShop, the browsers, etc. From a more technical point of view, I like to look at the various toolsets, libraries, frameworks, etc. but I really don't like to recommend one over others. I have a similar reluctance when it comes to recommending compilers (and books). There are so many good ones and which is best depends so much on who you are and what you are trying to do. Mentioning just one is unfair to the many good ones left unmentioned and mentioning all is impossible and would be tedious.

I guess that it is worth mentioning that I really like the standard library – especially its containers and algorithms parts ("the STL"). Some people still underestimate it "because it is not object oriented." For C++0x it's even better.

16) "Thinking in C++" by Bruce Eckel was one (probably the first) of the first full books available online for free on your language, did you read/take a sneak peak on it? and what did you think about it when you did listen about the first time?

I looked at Bruce's books long before they went on the web. I've known him since the 1980s before he even wrote his first C++ book. They were among the better C++ books when they were written. I suspect they still are.

17) I suggest always to learn another language and quite often I answer or a lower level or a higher level language; in an interview you answered "A useful language is a solution to a well-understood set of problems rather than simply something that fulfils all the currently fashionable criteria for what a programming language should look like", your was an problem-focused answer, are you still of this advice, or now something changed?

No, my opinion and approach have not changed on this point. A programming language exists to help you express solutions to problems. Thus, to learn or evaluate a language, you have to have a set of problems that you are interested in and look at the programming language in the light of those problems. Languages are not "good" or "bad" in isolation.

18) Which question is the one that journalists/programmers have asked you again and again in these years and you will avoid with pleasure?

"What do you think of 'language X'?" for some X such as C, C#, Haskell, Java, Perl, Python, or Ruby. Typically, the questioner hopes for some kind of controversial answer.

If I don't answer, I'm seen as evasive (obviously, I do have opinions and usually experience with the language in question). If express criticism, I'm perceived as rude or insufficiently expert to express a negative opinion. If I say something good about the language (there is something good about every language in significant use), I am reported as having said that "it is better than C++." And if I mange to give a polite detailed answer, some deem me obsessed by that language. Obviously not everyone make such interpretations, but enough do for me to dread that – inevitable and natural – question.

19) Have you some suggestion, advise or message for our Italian readers (any kind of message, it's up to you)?

"See above;" especially the answer to question 12

Really thanx,
Andrea Leganza