Interview with Lukasz lopuszanski for The Software Developer's Journal: About C++ and... few more things. March 2011.

C++ is undoubtedly one of the most successful programming languages of our era. I know You heard this question many times but today, looking from almost 30 years perspective, could you say what are in your opinion the greatest advantages and disadvantages of this language? From today's perspective what would you change/redesign in C++? :-)

➔There are two ways of approaching this question:
(1) If I had a time machine and could go back to 1979 and start over, what would I have done differently?
(2) What would I like to change today?

The time machine question is intriguing, but pure fantasy. If I had that machine, should I also go back a few years further to give some advice to Dennis Ritchie? Also, whenever I think of changes in the past I come up with a paradox: I know what happened, but 1980s-vintage Bjarne knew the local conditions in which C++ grew up better than I do (I have forgotten most) and he was almost certainly smarter than I – people rarely get smarter when they get older, just more experienced and knowledgeable.

On the other hand, what can be done today – given the constraints of billions of lines of C++ code, millions of programmers, seriously vested interests in tool chains, and committee processes – is so tempered by real-world concerns that the ideas easily gets lost in the technical details.

You can read serious answers to "what happened and why and what do I think about that?" questions in my two papers from the ACM History of Programming Languages conference (HOPL). They are available from my publications page.

What about C compatibility? Is C compatibility a good for C++? I decided to build C++ on the foundation of some existing programming language. Simula67 set an example by building on Algol and I really didn't want to re-invent the wheel and make the usual set of beginner's mistakes. I needed to build on a systems programming language because my aim was to deal with the increasing complexity of system programs arising from the huge increase in processor speeds and memory capacities. C seemed a good choice and at the time it was not as obvious a choice as it seems today. Many C and C++ programmers don't see it that way, but I think that C and C++ grew up together and both drew strengths from the large community.

From C, C++ got its direct and efficient model of the machine, which I think has been and is essential for its success as a systems programming language. That is something I would not change in retrospect. From C, C++ also got the messy declarator syntax, the error-prone narrowing conversions, and arrays with array-to-pointer conversions that seriously complicate range checking. These three features would not be in my ideal language. The idea of declarators is sound enough, but the syntax is an artifact of ancient parser technology and the Draconian demands of really small memories (48K bytes, if I remember correctly). A good syntax would not be one you could confuse with expressions and would be linear so that you don't have to use parentheses. For example

function f : (int, const string&)->(int)->char*;

That is a function called **f** taking a **(int, const string&)** pair of arguments and returning a function taking an **int** and returning a **char**. The suffix return type notation in C++0x takes a baby step in this direction.

Similarly, the narrowing conversions (e.g. **double** to **char**) are a historical accident and C++0x takes a baby step to help the programmer avoid the resulting problems. For example:

```
int x1 = 7.9;   // OK: c1 becomes 7
int x2 ={7.9}; // error: narrowing not allowed
```

The new uniform initializer syntax based on {}-lists does not allow narrowing. It also generalizes the arbitrary length initializer lists from C's array initializers to a general facility that can be used for user-defined types. For example:

```
vector<string> lang = {"BCPL", "C", "Algol", "Simula", "C++"};
```

The other key to C++'s success is its general and efficient abstraction mechanism. Classes basically works by simple aggregation of features and constructors and destructors is the key to many of the most effective C++ programming techniques, such as "Resource Acquisition Is Initialization" for structured resource management and effective use of exceptions. In particular, I do not regret avoiding a "richer set of abstraction mechanisms based on massive run-time support." I still consider that unsuitable for serious systems programming.

Template is an essential part of C++'s abstraction mechanisms, but here I do have an idea that – if it had been understood at the time – might have simplified both the language and its use. I think that a template design centered around "concepts" (sets of requirements for template arguments) would have been feasible, but I didn't know how at the time. I knew the problem and devoted the first three pages of the template chapter of "The Design and Evolution of C++" to it, but couldn't solve it under my demanding requirement for the template design. Maybe C++1x will finally get "concepts."

Would I have provided garbage collection? My ideal is to avoid garbage collection because it complicates local reasoning and can easily impose unsuspected overheads. Therefore I would not let any basic part of the language depend on garbage collection. However, as long as it is possible to leak resources in a language someone will do so. Therefore, even though I prefer the more general resource management techniques based on constructors and destructors, I would like to see optional garbage collection. In particular, C++0x provides an ABI for plugging in a garbage collector, but does not require anyone to actually use it. Please remember that memory isn't the only resource that a programmer can leak. Other examples are file handles, sockets, and locks.

Works on C++0x are still in-progress but the date of finalizing it, is not known. Why does it take so much time? What slows down the standards committee the most?

➔ How about May 16, 2011? That's the most likely date for the final technical vote on C++0x.

ISO standardization takes so long because it requires consensus among many stakeholders, because the requirements for the specification is tougher than for a language with a single implementer, and because the committee has no resources beyond the time of its volunteer members.

The lack of resources is the primary reason. Had we had a full time "design and experimentation" group to work on the specification, try out ideas, and coordinate community efforts, we might have finished five years ago. However, we don't – even I don't work full time on C++ design. The committee doesn't have a single penny to spend. Given these constraints all ISO standards takes many years to complete.

It is also worth remembering that the standard itself is something like 1,300 pages of dense technical text. It is a massive piece of work – and before you start to complain about language complexity, please note that other ISO language standards are of the same order of magnitude and we cannot achieve simplicity by throwing away facilities; that would break user code.

What is your estimation for finalizing the C++0x standard? And, maybe even more important: when in your opinion C++0x can become a standard in a practical sense (which means that it will be fully implemented by a compiler vendors and widely used by a C++ community)?

➔ I think we'll even get C++0x ratified in 2011. The implementers are already working hard on the new language features and essentially all of the standard libraries are shipping. You can start experimenting with C++0x now and you should because you really need to familiarize yourself with new features before trying to ship products with them.

The GCC progress seems to promise a large degree of completeness around the end of 2011. Microsoft is also moving ahead (e.g. you can download a free beta with lambdas, auto, and concurrency support), but I can't guess when any team of implementers will complete. If you ship with one compiler, there will be opportunities next year, but if you – like me – prefer portability across several compilers for important code, then you'll probably have to wait for another year. In all cases, I think it will be wise to try to adopt features in stages and try not to go wild trying to use as many features as possible right away. As ever, the aim must be "good code" not "maximal use of cool features."

Please note that even in this longish interview, I cannot explain more than a few C++0x features and I cannot go into details. If I refer to a facility without explanation, please consult my C++ FAQ for a more complete explanation. There, you can also find some overview, some discussion of design aims, and links to papers about C++0x.

C++0x introduces several new features to the language and its standard library. Could you describe five features which are in your opinion the most important ones?

➔ That's hard. I usually classify the improvements:

- Concurrency support
- Improved support for generic programming
- More and better libraries
- Better support for teaching and learning
- Etc. features

The idea is to indicate that individual features take their place in a more general approach to improve the support for programming styles. Picking a feature out of such a context can be quite misleading. Some of the smallest features are essential for the realistic use of features that appear much more significant to programmers.

I guess I could pick an example (the shortest example I can think of) from each of those general categories:

- Concurrency support: **async()** and **future**
- Improved support for generic programming: **auto**
- More and better libraries: **regex** (regular expression matching)
- Better support for teaching and learning: >>
- Etc. features: move semantics

We no longer have to remember to add a space at the end of a container of a container. For example:

```
vector<map<string,double>> v;
```

We don't have to specify the type of a variable if we initialize it. For example:

```
auto x = 7;             // x is of type int
auto y = v[i]["foo"]; // y is of type double
```

We can use a regular expression to find patterns in a file:

```
int main()
{
        ifstream in("file.txt");        // input file

        boost::regex pat ("\\w{2}\\s*\\d{5}(-\\d{4})?");    // ZIP code pattern

        int lineno = 0;
        string line;    // input buffer
        while (getline(in,line)) {
                ++lineno;
                smatch matches;      // matched strings go here
                if (regex_search(line, matches, pat))
                        cout << lineno << ": " << matches[0] << '\n';
        }
}
```

This is stripped of error handling code, but basically it looks for postal codes, such as **TX77845** and **NY 10027-7003**. The key function is **regex_search()**: **regex_search(line, matches, pat))** looks for the regular expression **pat** in the string **line** and deposit matches in the structure **matches**.

We can request tasks to be executed concurrently:

```
double accum(double* b, double* e, double init);          // some work function

double comp(vector<double>& v) // spawn up to 4 concurrent tasks
{
        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()** launches four tasks which may be run concurrently if the machine has suitable resources. Each **async()** returns an object (of type **future**) from which the result may be obtained using **get()**. The most important point here is that the programmer need not utter the words "thread" or "lock" to get the work done.

We no longer have to play around with references, pointers, clever tricks, or garbage collectors to return large objects from a function:

```
vector<int> make_test_sequence(int n)
{
        vector<int> res;
        for (int i=0; i<n; ++i) res.push_back(rand_int());
        return res; // move, not copy
}
vector<int> seq = make_test_sequence(1000000);          // no copies
```

Note that the (large) **vector** is returned by value. This is as efficient as it is simple because **std::vector** has a "move constructor" in addition to the usual copy constructor. A move constructor does not copy elements; it simply swaps representations leaving an empty object behind. Preferring the move constructor over the copy constructor for a return value that can never again be used is obviously a good idea. No special compiler magic is required; you can add move constructors to your own large data structures using a language feature called "rvalue references."

To give you an idea of the magnitude of work that has gone into C++0x, here is an incomplete list of useful new features and libraries:

- atomic operations
- **auto** (type deduction from initializer)
- **enum class** (scoped and strongly typed **enum**s)
- copying and rethrowing exceptions
- constant expressions (generalized and guaranteed; **constexpr**)
- **decltype**
- defaulted and deleted functions (control of defaults)
- delegating constructors
- extern templates
- suffix return type syntax (extended function declaration syntax)
- in-class member initializers
- inherited constructors
- initializer lists (uniform and general initialization)
- lambdas
- **long long** integers (at least 64 bits)
- **inline namespace**
- null pointer (**nullptr**)
- range **for** statement
- raw string literals
- rvalue references
- Compile-time assertions (**static_assert**)
- template alias (**using**)
- unicode characters
- user-defined literals
- variadic templates
- **std::array**
- **std::async()**
- **std::function** and **std::bind**
- **std::forward_list** a singly-liked list
- **std::future** and **std::promise**
- garbage collection ABI
- hash_tables (e.g. **std:: unordered_map)**
- metaprogramming and type traits
- random number generators
- **regex**: a regular expression library
- scoped allocators

- smart pointers: **std::shared_ptr**, **std::weak_ptr**, and **std::unique_ptr**
- threads (**std::thread**)
- Time utilities
- tuple
- system error

You can look up these features in my C++0x FAQ and elsewhere.

Could you provide some comment about removing Concepts feature from C++0x? Is there any chance that Concepts will be added to C++0x in the future? Many of its elements assumed implementation of Concepts.

➔ It was most disappointing to have to remove "concepts" from the set of features provided by C++0x. Concepts were to have been the central improvement of the support for generic programming, providing a type system for types and through that better specification of a template's requirements on its arguments, better error messages for template code, and better overload resolution for templates. For example, we could specify the standard library algorithm **find** like this:

```
template<InputIterator Iter, class Val>
        requires Comparable<Iter::value_type,Val>
Iter find(Iter b, Iter e, Val x);
```

In other words, **find** takes a pair of input iterators and a value of a type that we can compare to an element of the input sequence.

That project failed partly because of its technical difficulty (I have POPL and OOPSLA research papers describing the design of concepts) and partly because there were some disagreement in the committee over the ideals for the design. The result was that the committee decided that "concepts" were not mature enough for standardization. I reluctantly agree. I think we will see a new and better concept design in a future C++ standard, but that will require fundamental work – we can't just "polish up" the C++0x design. Doing that research and development work will take years, but we should get a facility that is simpler, easier to use, and cheaper to compile than what the C++0x design offered.

Do you take under consideration adding experimental libraries (ex. Boost) to the C++ standard? I'm not talking about adding single libraries but a way of easily extending the standard library (ex. Python has such set of libraries), If yes, how duplicates will be removed? (std::tr1::bind in boost::bind)

➔ Boost was conceived as a testing ground for libraries. However, boost libraries have no official standing so there is no problem "removing them." TR libraries, however, do have official standing so we are very careful to

(1) include only libraries that we are pretty sure are sufficiently widely useable to be part of the standard
(2) point out that they are not completely part of the standard, so they may change if/when they are included
(3) try to make transition to the standard as easy as possible.

I expect work on selecting a new set of candidate libraries in the form of a second libraries TR will start very soon after we ship the standard.

Once a TR has been absorbed in the standard, incompatibilities with the TR become the domain of backwards compatibility compiler options and the like. As far as the standard is concerned, the TR no longer exist.

What constructions can be use in C++ to support MMP platforms? Will the lack of constructions supporting multithreading be an obstacle in the future?

➔ What "lack of constructions supporting multithreading"? C++0x offers

(1) A memory model devised for modern (multi-core) hardware
(2) A set of types and operations supporting lock-free programming (e.g. atomics)
(3) A **thread** class
(4) A set of **mutex**es, **lock**s, and condition variable suitable for traditional work with threads
(5) An **async()** launcher and a **future** type for slightly higher level concurrent programming

This provides a typesafe set of facilities for conventional threads-and-locks programming. That style of programming may be the worst way to exploit concurrency, but it is universal on conventional systems and to be usable as a systems programming language C++ must support it well – so C++0x does. By "type safe", I mean no more **void****s, casts, macros, etc.

In addition C++0x provides **async()** and **future** as examples of what kind of higher-level model can be built from the C++0x foundation and the lock-free programming facilities for people who want to provide something radically different.

Usually C++ was considered as language for operational system programming and was widely used on mobile platforms (ex. Symbian OS). Do the Android system and the virtual machine Dalvik are sign of C++'s dawn as a standard system language.

➔ My standard description of C++ has been "a general-purpose programming language with a bias towards systems programming." Using C++ as the implementation language for infrastructure has been one of its major strengths forever and may indeed even be on the upswing.

C++ has unique strengths when it comes to building infrastructure and resource-constrained applications. Those strengths come directly from C++'s simple and direct mapping to the machine and its facilities for providing flexible and cheap (in terms of code size and execution time) abstraction facilities.

What should be the characteristic of programming languages in the future?

➔ I expect to see many programming languages in the future, just as we have many today. I do not think there will be characteristics that will be common to them all. Different languages will still have different design aims and different user communities. For the systems programming area, I hope to see greater type safety (perfect type safety may be possible, except for specific direct manipulation of memory, and would be very nice) and higher level concurrency models (note the plural; I don't think that a single model suitable for all uses is achievable). I think we'll see increased support for local reasoning and a decrease in the use of non-local information.

How in your opinion the future of the C++ will look? Do you think C++ has its place?
Is there any road map for C++ beyond the C++0x?

➔ C++ will hold its own as the premier language for development of infrastructure and the implementation of resource-constrained applications. I suspect that this formulation actually describe an area of increasing size and important in the computing world.

The standards committee does not have a road map for the future. It is extremely difficult to make long term plans and set longer term goals for a language controlled by a self-selected committee of volunteers. That will, of course, not stop me and others from trying, but there is nothing significant to report at this point. Whatever is done, stability of the definition of the language is a premium. ISO committees cannot proceed to break lots of existing code. There are billions of lines of C++ code "out there."

In high level programming languages (C#, Java, Python) you can see the influence of functional languages. Do you think this pattern will also apply to C++ in the future?

➔ I guess that it is worth pointing out that in areas C++ is a higher-level language than those you mention. In particular, its abstraction facilities (as represented by the C++ static type system) are more powerful. More specifically, the STL brought many functional-programming techniques into the mainstream and C++'s templates is a Turing-complete functional programming language. The improvements to C++'s support of generic programming (such as **auto**, **decltype**, variadic templates, uniform initialization, and lambdas) implies better support for functional programming techniques.

That said, we must be careful to learn from the strengths of functional programming without getting caught up in its problems. Functional programming in various forms has ruled academia and large sectors of the educational establishment for decades, yet its impact on delivered systems has been very close to zero.

There are a lot of high quality resources (books, articles, etc.) for learning C++98. C++0x misses these. Where C++ programmers can look for such resources? What in your opinion is the best way to learn C++0x?

➔ It's a bit early to expect good educational support for C++0x; even C++98 is not over-endowed with great teaching materials. First, we'll see descriptions of individual C++0x features. My C++0x FAQ is a good example.

What will take longer are coherent descriptions of how to write programs using a balanced set of language features and standard library facilities to write comprehensible, well-performing, and maintainable code. We need to reach a level of understanding where we don't consider language features in isolation and don't see C++0x as a layer of features on top of C++98, but as a new language with a coherent and integrated set of features and standard libraries.

That will be a tall order, because many still insist seeing C++ as a layer on top of C. That is an approach that leads to sub-optimal and unmaintainable code.

As an example of an modern, integrated approach to learning C++ (just C++98, unfortunately), I can recommend my new textbook for beginning to medium programmers: "Programming: Principles and Practice using C++." There, libraries are used early and low-level C-style features are not discussed until Chapter 17. On the other hand, it is not a "OO from day one approach": classes and class hierarchies are not introduced before they are needed for realistic problems.

Are you working on The C++ Programming Language, Fourth Edition?

➔ Yes, but it is a lot of work so it is still more than a year in the future – maybe even much more. The problem is that I must explain how to use the language and thinking that through takes time. I have no desire just to document the features or even to explain each feature in isolation. That's what (online) documentation is for and you already have my C++0x FAQ providing a lot of that. I need to explain the rationale for the various facilities and show their use in combination in support of programming styles.

Also, I find it hard to find the many days of uninterrupted time that is required for that kind of thinking, experimentation, and writing. My day job requires attention and C++'s continuing success results in many demands on my time.

How have the skills required for programmers changed since you designed C++?

➔There is a greater need to understand the type system and the benefits of using the type system well has steadily increased. C++0x is a further development in that direction and the richness of the static (compile-time) type system is one area where C++ differs from most other popular languages.

Looking beyond the programming language, the changes in software development have been dramatic. In 1980s terms, modern PC-application programs are "incredibly huge," "unbelievable complex," and "unbelievable wasteful of resources." Many programmers do nothing except finding the appropriate corner of a framework to exercise. That's not the kind of programming that I'm most interested in or for which C++ shines. I focus more on software infrastructure and resource-constraint applications where C++ has most to contribute. A lot of interesting embedded systems fall into these categories. For examples of C++ use, see my applications page.

And I have to ask this question :-) What do you think about modern programmers, are they better or worse?

➔Some are better and some are worse. A relatively small "elite" still rely on extended academic education, massive technical experience, and a certain amount of idealism to actually understand the systems they work with and try to further the state of the art. There may indeed be more such programmers working today than ever before. Unfortunately, they are often overlooked and drowned out by a larger number of people willing to just patch a corner of an incompletely understood system just to ship something – anything – on time. Of course I'm caricaturing a bit, but I really would like to see more idealism and professionalism in the software field – and both starts with a thorough grounding in the fundamentals of computing.

Please note that this is (emphatically) not a statement that all software should only be written by PhDs from top universities using mostly Greek letters. It is meant as an encouragement to try to understand the underlying principles of every system you work on and to aim for demonstratively correct code, rather than just memorizing which switches to set to get a desired output and hoping that bugs turn up in the testing.

Do you have any "tips" for Polish programmers?

➔ Know your fundamentals (algorithms, machine architecture, and systems) and a couple of programming languages well enough to write idiomatic code in them. You can write Fortran or C in any language, but that's typically a misuse of the language. Be wary of fads and ever so often do something just because it's different. Work on your (verbal and written) communications skills. Poland has a proud history in math and engineering, try to add to it.

On a lighter note, please note that my programming textbook for non-experts "Programming: Principles and Practice using C++" just came out in Polish. It's in C++98 and would have been much better if I could have used C++0x, but I still think it provides a superior approach to learning programming of the kind needed for the application areas in which C++ is used.

Thank You for this interview!