This interview has four main sections

1. C++0x from now to the release of the Standard.
2. post C++0x future evolution
3. other programming languages
4. how to teach C++

C++0x from now to the release of the Standard.

1. Is there any big change that may be committed to draft from now?
2. Is there any big issues left in current draft?
3. What benefit does C++0x offer, and what feature you think it's important, for beginners, average users, and advanced users?
4. When exactly will the Standard be released?

Q: Is there any big change that may be committed to draft from now?

BS: We just (March 2010) produced a final committee draft (a FCD), which is currently being reviewed by the national standards bodies and voted on. No big changes are supposed to happen from now on and I don't think that any that I would consider major will happen. We can all think of features and libraries that we would have liked included. For example, I would really have liked to see concepts and a more uniform function declaration syntax in C++0x. These features were so close to ready! I would also have liked to see a version of the boost file system and asio libraries included. However, if we had to wait for every feature I and others might like, we'd never get a new standard.

I think that – except for many little clarifications – the standard is ready to ship. Obviously, we could wish for much more, but that will have to wait for the next round of standardization. Compared to C++98, C++0x is a significantly better tool for writing good code. By "good", I mean "correct, comprehensible, efficient, and maintainable."

Q: Is there any big issues left in current draft?

BS: No, not really. Naturally, there are facilities that C++0x does not provide and issues that it does not address. That is true for every language, but I don't see a new facility offered that has major problems with the form in which it is offered. Since no language is perfect, I fully expect that users will find problems using some new features. However, my guess is that the most typical problems will relate to learning to use the many new facilities (some people will complain of too much novelty) and to compensate for lack of hoped-for facilities (some people will complain about lack of new features).

It is worth remembering that many in the C++ community still have not completely caught up with C++98. I actually hope that C++0x will help those people by making modern programming technique (beyond the C-style, OOP-style, and early over-elaborate generic programming styles) easier to learn and apply in real-world situations.

Q: What benefit does C++0x offer, and what feature you think it's important, for beginners, average users, and advanced users?

BS: You can find lists of features in my C++0x FAQ and also examples of almost all new facilities. However, I don't think that a list of language features gives a good view of what C++0x offers to a programmer. The important point is that the new and old features work together – and together with the standard library facilities – to make it easier to express good design ideas. Think of a language feature as a kind of brick in a building set. What matters is not the individual brick, but how the bricks can fit together to allow for a huge variety of constructions. I have tried to give an idea of what becomes possible in my 2009 ACCU paper "*What is C++0x?*" You might find that a more useful introduction than the C++0x FAQ and much much better than the FCD text itself. The (daft) standard is written by and for experts with little concern for tutorial aspects. The "heavier" technical documents should ideally be read after the tutorial material.

As an example, consider the old problem of returning a large value from a function. For example, a matrix add needs to return a matrix:

```
Matrix operator+(const Matrix&, const Matrix&);
```

Given that, we can write something like

```
Matrix a = b+c+d+e+f+g;
```

In C++98 – unless you were unusually clever in your design of the **Matrix** class and the **operator+()** – you would have a major performance problem on your hands. You would have to copy **Matrix** objects to get results out of the **operator+()**. If a **Martix** was large, e.g. a 100-by-1000 matrix of doubles, the cost could easily make such code impractical for real-world uses. Often, people try to address such problems by returning a pointer or a reference and then they end up with non-trivial memory management problems. This is one kind of problem that can push people towards a garbage collected language for reasons of simplicity and performance.

C++0x offers the notion of move constructors and move assignments as a solution. Consider a simple matrix class:

```
class Matrix {  // 2D matrix
        double* elem;           // elements stored on free store
        int dim1, dim2;         // dimensions
public:
        // …
        Matrix(const Matrix&);                  // copy constructor: copy elements
        Matrix& operator=(const Martix&);       // copy assignment: copy elements

        Matrix(Matrix&&);                       // move constructor: don't copy elements
        Matrix& operator=(Matrix&&);            // move assignment: don't copy elements
};
```

The move operations are new in C++0x. The compiler will use a move rather than a copy if the object copied from isn't needed any more. The **Matrix** move constructor is trivial:

```
Matrix::Matrix(Matrix&& m)
```

```
{
        elem = m.elem;          // copy the Matrix representation
        dim1 = m.dim1;
        dim2 = m.dim2;

        m.elem = 0;             // clear the representation of the argument Matrix
        m.dim1 = 0;
        m.dim2 = 0;
};
```

In other words, we don't copy the elements, we just transfer the ownership of the elements from the source to the target. That's 6 simple assignments, rather than 100,000.

Given the **Matrix** move operation, defining an efficient **operator+()** is trivial:

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
        if (a.dim1!=b.dim1 || a.dim2!=b.dim2) error("unequal dimensions in Matrix +");
        Matrix res(a.dim1, a.dim2);
        for (int i = 0; i<dim1; ++i)
                for (int j = 0; j<a.dim2; ++j)
                        res.elem[i][j] = a[i][j]+b[i][j];
        return res;
}
```

The point is that the trivial and obvious implementation of the **Matrix** addition has become efficient. The compiler knows that **res** is about to be destroyed upon return from **operator+()** so it moves **res**, rather than copying it.

A typical user needs only know that a matrix can be moved efficiently. The technical details can be left to expert users and library implementers. A move constructor takes an argument of "rvalue reference type" indicated by the **&&** notation. An rvalue reference is a kind of reference to which an rvalue can be bound. Since a move constructor needs to modify its argument to move the information from it (to transfer ownership of memory or other resources), that rvalue reference is not **const**. So, for a class **X**, the move constructor looks like this

```
X(X&&);         // move, rather than copy
```

For more details, see the "What is C++0x?" paper.

Note that move constructors is not an esoteric feature of interest only to people writing or using advanced numerical applications. Needing to return a large object (e.g. a set of results from a queries) or an object that cannot be copied (e.g. a file handle or a lock) is common in many important areas of programming. Given move constructors (and move assignments), we can now handle those efficiently in the most obvious way without having to resort to the use of pointers, references, free store, or clever management schemes.

I chose move constructors as my first example of C++0x because it addresses an important class of real-world problems, does so in ways that improve performance, can be used by novices, and is rather innovative. Basically, move semantics can dramatically simplify C++ code while improving its performance.

Follow-up question: But is your Matrix example efficient enough?

Your operator+() allocates a lot of memory for elements; can't you just reuse elements from arguments?

BS: The **Matrix** example is an example of a general technique for returning large values. Often, optimizing a return is exactly what is needed and therefore efficient enough. However, for the matrix add example we can – as you indicate – do even better by also optimizing the way we receive arguments. For example, if the second argument is an rvalue, we can re-use its elements for the result:

```
Matrix operator+(const Matrix& a, Matrix && b)
{
        if (a.dim1!=b.dim1 || a.dim2!=b.dim2) error("unequal dimensions in Matrix +");
        for (int i = 0; i<dim1; ++i)
                for (int j = 0; j<a.dim2; ++j)
                        b.elem[i][j] += a[i][j];
        return b;
}
```

This saves us from having to construct a local variable to hold the result. Of course, to do this the implementer needs to know about rvalue references, but again the user can be oblivious.

In general, it should be noted that for industrial-strength high-performance computing no single optimization is sufficient. Designers and implementers of such libraries must – in addition to being domain experts – know a lot about the language and typically also about the optimization used on various implementations. The strength of C++0x in this context is that an implementer can do much better than in most languages with simple and portable techniques.

What else does C++0x offer? We can classify it like this:

- Support for concurrency (memory model, threads, locks, futures, the task launched async(), and more)
- Support for generic programming (uniform and general initialization, variadic template arguments, lambda expressions, type deduction from initializers, template aliases, and more)
- Better facilities for defining and using classes (inhering constructors, control of defaults, move semantics, user-defined literals, and more)
- Etc. language facilities (strongly types enumerations, general constant expression evaluation, a range-for loop, C99 facilities, compile-time assertions, a null pointer, and more)
- Standard library components (regular expressions, resource management pointers, hash tables, random numbers, garbage collection ABI, type traits and metaprogramming support, and more)

Unless you already know something, this is just a collection of buzz words. I hope you find some of those words tempting enough to explore their meaning. All the new facilities were designed to address problems considered to be serious and most of the work was in integrating the features into the language so that they could be used in combination. It is hard to classify features because in different combinations they are part of solutions to different problems. Many features can be described as improving C++ as a tool for defining and using "light-weight" (i.e. simple and efficient) abstractions.

Let's consider the second part of your question "what feature you think it's important, for beginners, average users, and advanced users?" With few exceptions, a feature that is important for a beginner is

important for all users and a feature that is important for an expert is indirectly important to a novice as it improves libraries. Consider:

```
vector<list<int>> vld = { {1,2,3}, {3,4}, {}, {9,8,7,4} };
```

Obviously, this is easier to write for experts and novices than any C++98 alternative: The standard-library containers support variable-length initializer lists (and assignment of such lists, and the use of such lists as function arguments, etc.).

Did you notice the absence of a space between > and > above? That was intentional and arguably the smallest improvement in C++0x. Not having to remember that space certainly helps novices, but it also helps me. I write a lot of template code – some of it complicated – and frequently forget that space.

The oldest C++0x feature is the use of **auto** to indicate that an object is to have the type of its initializer. For example:

```
auto x = 2.3;     // x is a double
auto y = 'c';     // y is a char
for (auto p = v.begin(); p!=v.end(); ++p)        // p is an iterator for v
        cout << *p << '\n';
auto z;           // error no initializer
```

The **auto** specifier gets rid of tedious typing of (sometimes complicated) type names without losing static type checking. This is the oldest feature because I implemented it in the winter of 1983/84, but had to take it out again because of a C compatibility problem. Again, this helps both novices and experts. In particular, **auto** saves experts from specifying complex types that depend on template arguments. For example:

```
template<class C> algo(C& c)
{
        for (auto p = c.begin(); p!=c.end(); ++p) // do something
}
```

In such cases, the use of **auto** can prevent error-prone and redundant repetition of type information.

There are of course also features that are primarily aimed at expert programmers, especially library writers. Obvious examples are some of the foundational support for concurrency, such as the atomic types for lock-free programming and the mechanism for transmitting a thrown exception from one task to another.

Q: When exactly will the Standard be released?

BS: My guess is 2011. I expect that there will be many comments to the Final Committee Draft, but not more than the committee can handle in two or at most three meetings. That would make the standard ready for ratification next year.

I think that the question that most people ask is not "when will we have a standard?" but "when can I use these new features?" You can begin to experiment with some new features today: For example, GCC is sipping many (in version 4.5) and Microsoft has some in their latest beta (available for free download).

Good approximations of the new standard library components are shipping widely and if your purveyor doesn't ship their own version, there is boost.

I do not know when a vendor will provide all C++0x features or when all the major vendors will provide all. This, for many users, will determine when they can ship products in C++0x. Tool builders may even have to wait until all of their users upgrade to C++0x compilers. This is going to take time – years – but everyone agrees that we are now in a better shape with the correctness and completeness of the specification than we were at the same stage of the 1998 standard and we have more complete implementation experience.

Post C++0x future evolution

Q: Do you have any idea about new features?

BS: I think it is a bit premature to speculate about future extensions before the standard has been completed, but that never stopped anyone – not even me. My first choices would be redesigned and improved concepts (basically type checking for sets of types – especially template type arguments), some higher level concurrency models on top of the basic threads and locks layer provided by C++0x, and several new standard libraries.

I also have a very nice proposal for multi-methods that I'd like considered: *Open Multi-Methods for C++*. Consider the old problem of determining whether two shapes intersect. We want a function

```
bool intersect(Shape& s1, Shape& s2);
```

The snag is that Shape is polymorphic. For example, **s1** may be a **Triangle** and **s2** a **Photo** of the Tokyo tower. We need a dynamic (run-time) lookup on *both* arguments. In C++ – as in all conventional OO languages – we need to do two dynamic lookups (two virtual function calls). With multi-methods we would write:

```
bool intersect(virtual Shape& s1, virtual Shape& s2);      // base function
bool intersect(Triangle&,Square&);                         // derived function
```

Assuming that a **Photo** is a **Square**, a call of the base function (the function taking arguments of the base classes) will invoke the correct derived function. The use of **virtual** on an argument indicates that the actual dynamic type of an object will be used to select the function to be called. Our measurements show that a multi-method lookup on two virtual arguments is less than 10% slower than a virtual function call and thus much faster than the two virtual function calls needed by the traditional solutions to these problems.

Note that **intersect()** is a free-standing function. Not requiring a multi-method to be a member of a class allows us to gain very useful functionality from multi-methods with only on argument. Consider the classical problem of providing operations on an abstract syntax tree. Since the set of operations cannot be fixed by the provider of that tree and the users of that tree cannot modify its declarations, programmers conventionally resort to the visitor pattern. However, I consider the visitor an inelegant workaround. With multi-methods, we simply define our operations as a hierarchy of functions "on the side" of the class hierarchy. For example:

```
void print(virtual Expr_node&);
```

```
void print(And_node&);
void print(Throw_node&);
// …
```

I would be willing to pay a lot to avoid having to write another visitor.

Please note that at this stage nobody even knows what will be seriously considered for the next standard. For now, anything beyond C++0x is just dreams and experiments.

Q: What features need to be removed or deprecated from the Standard?

BS: For C++0x, we deprecated exception specifications and removed **export** (separate compilation of templates). The export feature never caught on and might in the future be compensated for by concepts. The exception specifications were more trouble that they were worth and are replaced by the much simpler (and more run-time efficient) **noexcept**.

It is really hard to remove something from a language in widespread use. Remove something significant and you make a few hundred thousand enemies by breaking their code. Remove something insignificant and you have done nothing worthwhile. All you can do is to try to be very careful about additions, try to simplify through generalization, and provide superior alternatives to old features that cause trouble.

In that vein, I hope to eventually replace essentially all unconstrained template arguments with properly constrained ones: To do that will take a redesigned, refined, and hopefully simpler notion of concepts.

Other programming languages

Q: What languages do you usually use?

BS: Mostly C++ plus a scripting language for quick simple tasks.

Q: What features/paradigms do you like?

BS: I think that the word "paradigm" is seriously overused and in most contexts ill defined. I try to avoid using that word. Most often, people imply some kind of exclusivity when they say "paradigm." They think that if they use object-oriented programming then they cannot also use generic programming and that somehow one paradigm is better than all others for all uses and all users. This is not so.

I appreciate a variety of styles of programming – such as OOP, GP, and traditional C-style programming – and use them in combination. I find that for most significant problems the best solution involves several such styles. Here is a classical example:

```
template<class C> void draw_all(const C& c)      // draw all shapes in c
{
        for (const auto& p : c) p->draw();
}

vector<Shape*> v { new Circle(p,20), new Triangle<p,p2,p3>, s3, rect0 };
list<Shape*> lst { new Circle(p,40), new Rectangle<p,p2>, sx3, rect10 };
```

```
draw_all(v);
draw_all(lst);
```

Obviously, this is object-oriented programming because I use a polymorphic type **Shape**. The example of drawing all objects from a container as an example of polymorphism goes back to the early presentations of Simula67. I also use generic programming to define my containers (**v** and **lst**) and traverse them using the C++0x range-**for** loop.

Alternatively, I could have used the standard-library algorithm **for_each** to emphasize that I'm using generic programming:

```
template<class C> void draw_all(const C& c)
{
        for_each(c.begin(),c.end(), [](Shape* p){ p->draw(); })
}
```

In this variant, I use a C++0x lambda function to define a function object to be passed as the third argument to **for_each**. The notion of lambda functions is borrowed from functional programming.

There was a time not so long ago where such combinations (and C++ in general) were derided as "hybrid" and "not pure." These days, most modern languages support some combinations of "paradigms." For example, O'Caml and F# combine object-oriented and functional techniques. Java and C# have adopted generics and a minimal amount of generic programming techniques. I think this is a good trend and a sign of maturity in our otherwise all-too-immature field.

Q: Is there anything you don't like?

BS: Marketing hype as a substitute for technical argument. Thoughtless adherence to dogma. Pride in ignorance.

How to teach C++

Q: I think details are important.

You said at the interview (http://www.devx.com/cplus/Article/42448/0/page/2)

> Most programmers need not go there—and in my opinion, far too many programmers agonize over obscure details, such as "what does void A::f()&&; mean?" and "how is std::move() implemented?" I try to understand and answer such questions only when I have a reason to. Most programmers could use their time better writing idiomatic code; we need good programmers far more than we need yet-another language lawyer.

I think many programmers DO agonize these questions. Do you really think users will satisfy just explain: "std::move() magically invalidates the object to move the value somehow. You don't need to understand it. Just use libraries."

> I think if one cannot understand a language feature, he won't use it. That includes the libraries which use that language feature.

BS: Thank you for letting my use such an extensive question. There are many levels of understanding. I don't understand much about Windows internals, yet I use Windows every day. I know a bit more about Linux internals because I once knew Unix very well, but I always have to rely mostly on trust, knowing that there are many things I do not know about the system I use and that I will never know those except at a superficial level. Today's software is far more complex than any of us really imagine and let's not even start explaining the marvels of hardware. Programmers who want to understand every feature of a language and every feature in complete detail is making a fundamental mistake. They cannot understand all that and they should not even want to!

I drive a car every day, but I don't want to really understand its engine. I do not want to become a car mechanic or an automotive engineer. Instead, I want to be – and I think I am – a good driver. For that I must have a good understanding of the controls of the car, of the rules for driving on public roads, and of the likely behaviors of other people and vehicles on the road. Knowing all about my car is not necessary and could even make me a worse driver by distracting me from my necessary driving skills. We need to adopt the notion of levels of abstraction to the task of programming. We need to understand a programming language at a level of detail and abstraction sufficient for using a set of techniques. An aim of understanding all and having that understanding available at all times is counterproductive. Even I can't answer every question about C++ without reference to supporting material (e.g. my own books, online documentation, or the standard). I'm sure that if I tried to keep all of that information in my head, I'd become a worse programmer. What I do have is a far less detailed – arguably higher level – model of C++ in my head.

What programmers should know is the basic facilities of the language, the basic of the functioning of the main features, and how to gain more knowledge as needed. In other words: People need a model of the language and have access to information sources. I do not require people to believe in magic. Never! There is far less "magic" in C++ than in other modern languages and I think that is part of the problem. You can look at a standard-library algorithm or a boost library and see exactly how it is put together. Sometimes, reading such code is an expert-level task. That scares people. For a language such as Java or C# you simply don't get to see the equivalent – it is (like the OS functionality) buried in components that average users never see or even in code that they cannot see.

To use the example, **std::move()** mentioned above: I can quite reasonably explain the use of move without explaining the implementation. Remember the move semantics I discussed for **Matrix** in the earlier question. There, the compiler knew to use move (rather than copy) when returning a **Martix** object because the compiler knows we cannot use a local variable after a **return** from its function. Sometimes, you know that a value won't be used again but the compiler cannot be assumed to kown that. In that case, you'd like a *move* rather than a *copy*. You tell the compiler to prefer a move by using **std::move()**. Consider the C++98 **swap()**:

```
template<class T> void swap(T& a, T& b)
{
   T tmp = a;          // copy
   a = b;              // copy
   b = tmp;            // copy
}
```

Here, we don't really want to do make any copies, so for C++0x **swap()** has been rewritten to say so:

```
template<class T> void swap(T& a, T& b)
{
   T tmp = move(a);   // move, don't copy
   a = move(b);
   b = move(tmp);
}
```

This **swap()** can swap **Matrix**es without copying because **Matrix** has a move constructor and a move assignment. Many users of **swap()** will experience performance improvements without ever having heard of "move."

Do you still really want to know how move is defined? OK, it is just a cast from lvalue to rvalue:

```
template<class T> T&& move(T& a)
{
   return static_cast<T&&>(a);
}
```

The standard-library **move()** allows the programmer to tell the compiler that a value is no longer needed. That can be valuable information to a compiler. Personally, I think that **std::move()** should have been named **std::rvalue()**, but you cannot change the name of something after years of use. Does it help to know that detail about **move()**? Do you become a better programmer for knowing that? I doubt it. It is important that you can gain an understanding of every feature you really want to know well, but to want to know every detail seriously distracts from becoming and effective software developer. As I said in that quote: We need good programmers far more than we need yet-another language lawyer.

Q: Isn't C++ too hard for average user?

> The reason many Japanese programmers don't use STL is because they can't understand the templates. You can argue it's for the library implementer, not for the user. But if most users won't use such libraries, it's worthless. You can argue we need a better education. But that didn't happen, at least in Japan. Remember, it was Japan which proposed EC++.
>
> How could I learn C++? I learned C in three months. Well, there still were many details I should had to know, but I can use C in three months. For C++, it took me so many years – mainly because of templates. I found STL is so useful. But I couldn't use it well because I couldn't understand templates. There were (and still are) no good Japanese book which explains the template. Most Japanese books just explaining how to implement min()/max() without using preprocessor macro and that's all. To learn templates, I had to read English books. Eventually I started reading Standard and here I am. Now I know how to write templates and many tricky meta-programming techniques which is used in Boost libraries. I can learn new C++0x features by reading the Standard wording. But this isn't a way to learn a language for most programmers. Most programmers can't spend years just to learn language grammars.

BS: No. Millions of people have learned to use C++ reasonably well. I would of course like for C++ to be easier to learn and for programmers to use it better, but you don't have to be a genius to write good C++.

Also, you can learn (only) some of C in three months. Similarly, you can learn (only) some of C++ in three months. I claim that if you chose your subsets of C and C++ well, you will be a more effective

programmer in three months if you chose C++. "Most of C" is not the ideal subset of C++ for early learning. I see two distinct scenarios:

(1) You are a non-programmer trying to learn to program and to learn your first programming language (C or C++) at the same time
(2) You already know how to program (in some language) and are trying to learn C or C++.

For (2) the difficulty of learning C or C++ depends critically on what the person already knows. If – as was common in the past – most people knew a language that was fundamentally similar to C in the way it was used. So to them, C seemed much simpler than C++. The new C user basically had to learn only a new syntax, whereas a new C++ user had to learn that plus something about the definition and use of classes and class hierarchies. Even then, C++ can be easier because the C-style subset of C++ is easier to learn and use than C: a person is offered a better a type system, better notational support for traditional styles of programming, a more extensive standard library, and needs to learn fewer workarounds. I discussed that in my paper "*Learning Standard C++ as a New Language*."

For non-programmers, it is important to keep the focus on programming rather than on language-technical details. This can be hard. I faced that problem a few years ago when I decided to design a new first programming course for electrical and computer engineers at Texas A&M University. The result was my book "*Programming: Principles and Practice using C++*" which is already being translated into Japanese. To write that book, I had to think hard about the relationship between language and the task of programming. I present language features primarily as supports for programming techniques. I also had to think hard about what should be emphasized early and about the order of presentation of topics. I chose topics to discuss primarily based on the needs of real-world programming, rather than ease of teaching or the ease of setting tests. The result that emerged after a couple of years of teaching differs both from the traditional bottom-up approach and the currently popular "object-oriented" approach. I use standard-library components from day one (string and vector) and limit my use of built-in types to the essentials (bool, char, int, double). The focus is on how to solve programming problems, not language-technical details. I present classical control structure and I/O techniques early on, but soon (Chapter 6) get to defining classes. Class hierarchies are not introduced until they are needed for a simple graphics and GUI library (Chapter 14). The introduction of pointers and arrays are postponed until much later (Chapter 17) where they are discussed in the context of defining data structures that can handle variable numbers of elements (e.g. vector). You can find a description of my approach in www.stroustrup.com/Programming and a discussion its intended role in a more general education in my paper "*Programming in an undergraduate CS curriculum*". I even have examples where this book – intended for beginners, programming courses, and self-study of programming – has been used by more experienced programmers who wanted to update their C++ knowledge to something more up-to-date. That may be important because I see a lot of views of C++ that seems stuck in the late 1980s.

Can you learn to use templates well? The answer is yes, and my opinion is based on the fact that most students can use simple templates (such as vector, map, and basic STL algorithms) and many can write simple templates – all by month three of an ordinary university freshman (1st year) course. Note that I do not claim that the students know all of the rules for C++ templates or are capable of understanding non-trivial boost components (e.g. the graph library); they are not. What we need most is not yet another book on how to write complicated template, but a good tutorial article showing how to write and use simple

templates well. Templates are essential for type-safe techniques in C++ and the backbone of time and space critical applications. The latter, in particular, is surprising to many. Templates can be overcomplicated and too many complications and too many template arguments don't just scare new users, they can also lead to unexpected and unnecessary overheads (e.g. see the OOPSLA'09 paper that I co-authored "*Minimizing Dependencies within Generic Classes for Faster and Smaller Programs*"). As in every area of programming, the ideal is simplicity. For some, my book "*The C++ Programming Language (3rd Edition)*" is a good introduction to templates (and that is available in Japanese).

It is not a secret that I am not a fan of EC++ (e.g., see my FAQ) and think that its definition was (at least partially) based on flawed assumptions. In particular, you do not necessarily make life easier for programmers by restricting them to a sub-set of a language. I think that coding standards is the right way to subset: In that case you can define a subset plus libraries that approximate the ideal for a specific application area (e.g. see the JSF++ coding guide for avionics). "Embedded systems" is far too broad an area to allow the definition of a single coding standard for all applications let alone a good restrictive language subset. My opinion is that the worst mistake in the design of EC++ was to leave out templates, thereby failing to support type-safety and efficient close-to-the hardware abstraction. The alternatives lead to more complicated and less reliable code. My impression is that where EC++ is used today, it is mostly EC++ plus templates plus another few facilities. My opinion is that (full) ISO standard C++ is a better choice – which can and should be supported by a good coding standard where needed.

The key to successful use of C++ "close to the hardware" is to view is as providing two aspects:

- C++ provides a simple model of real hardware (mostly shared with C)
- C++ provides flexible mechanisms for building efficient (in time and space) abstractions

For some details see my paper "*Abstraction and the C++ machine model*." For even more details, see the standards committee's "*Technical Report on Performance*".

I hope that C++0x will help many programmers to use C++ better. It allows better expression of ideas and being new, it may inspire people to re-examine their assumptions about C++. Maybe C++0x will inspire people to write tutorials emphasizing simple use, rather than just papers showing off cleverness.

Follow-up question: Speaking of "coding standards," isn't this the same?:

1. The subset standard which doesn't have templates.
2. The coding standard which forbid the use of templates.

> Sometimes, coding standards makes problem even worse. I've heard many silly coding standards which forbid the use of templates, the STL, and the Boost libraries. Not because of technical reason, but because some non-programmer can't understand these features.

BS: A coding standard should be designed with care for a well-defined application (or application area); otherwise, it does harm. Too often, coding standards are devised by people with weak C++ experience who are overly influenced by fears of novelty. In particular, this happens when an organization moves from a different language to C++. I have seen cases where an individual's first job with C++ – before even writing a single significant C++ program – was to write a corporate coding standard. In such cases, people often try to ban features that weren't present in their previous language and also try to enforce rules devised to avoid problems in that previous language. This can seriously harm C++ use. For example,

people coming from C have been known to be very afraid of function overloading, banned it, and then found that they had to use a lot of (highly error-prone) macros to simulate generic programming. That's counterproductive. Another example is a ban on doing "complicated stuff" in constructors, leading to (needlessly error-prone) two-phase construction and misuse (or underuse) of exceptions. The resulting C++ code looks as if it wasn't written by competent C++ programmers. Use of Hungarian notation in C++ is another obvious example.

Also, a good coding standard is not just (or even primarily) a set of prohibitions. A good coding standard is prescriptive: it gives guidelines for effective use of the language. I think that is best done by recommending the use of library facilities to allow people to avoid error-prone and unproductive uses of the language. The basic philosophy of the JSF++ coding standard (which I formulated) is that *first* you extend the language with library facilities suitable for the application area, *then* you subset the language used directly by application builders. Doing so allows you far simpler and far safer rules that a simple "subset the language" set of rules. For example, the JSF++ deals with the host of pointer conversion, pointer/array, and array overflow problems by a simple rule: No arrays in interfaces (e.g., as function arguments). To pass collections of data, it provides a couple of simple containers (which do not offer unsafe implicit conversions and "knows" their number of elements). A coding standard that tries to enforce safe use of pointers (e.g., MISRA) needs dozens of rules does nothing to improve programmer productivity, yet still leaves opportunities for serious problems. I consider such problems inherent in the simple subsetting approaches: they leave the programmer with serious problems inherent in the language but without the language facilities provided to cope with those problems. For a C++ subset, this typically implies the programmer having to deal directly with low-level language features (e.g., pointers and casts) without the help of the higher-level abstraction features (e.g., classes and templates). That's error-prone and unproductive.

A fundamental difference between a language subset and a subset defined by a coding standard is that the complete ISO C++ is available to the implementers of the infrastructure for the coding standard: a good coding standard is a subset of a superset. That superset is C++ plus application specific library components.

I consider templates fundamental to type-safe and efficient C++. For example, unless you can restrict yourself to a single element type, you need containers and to implement type-safe efficient containers (and operations on such containers) you need templates. The built-in array is at best a poor substitute for an appropriate type-safe container. I could imagine a coding standard that prohibited many uses of templates by application builders, but not a good coding standard that prohibited their use in the application software infrastructure. However, I have never seen an application area where I would prohibit all uses of templates by application builders. On the other hand, every coding standard I can imagine would try to limit template use to what is simple in that application area.

Q: But how do a programmer decide what's important to learn? What are the basics we need to learn and what are the complicated parts that we should avoid?

BS: That's a very hard question. If you look at deployed code (e.g. boost libraries, open-source projects, or Windows applications), you see code that is written – often under time pressure – with little concern for pedagogical issues. Reading them assumes knowledge of a huge part of the language. Often the code written in complex ways to work with outdated compilers, to interact with libraries with interfaces which were not designed with C++ in mind, to optimize performance, and to be portable across a large set of

hardware and operating systems. Reading such code is not good for initial learning. Often, library source code is far more complex than application code.

Another approach is to rely of simple web tutorials and/or a vendor's online documentation and examples. The problem with that is that you tend to miss out on understanding of the key concepts and techniques. Using that approach, you tend to learn (good and bad) by rote. Some people can generalize from that, but many keep mindlessly repeating exactly what they first saw because they don't understand why it works and don't dare trying anything different.

In my opinion, there is no substitute for a good traditional textbook. It is the job of a textbook author – not the student – to decide what is important and what is not. A good textbook does not simply present everything and does not give equal emphasis to everything it presents. Furthermore, the order in which topics are introduced is important for a reader. That's easy to say, but I have worked hard over the years trying to make that theory reality. "*The C++ Programming Language*" and "*The Design and Evolution of C++*" are available in Japanese and "*Programming: Principles and Practice using C++*" will soon be. The papers I mentioned above (all available from my home pages) are more specialized, but can also be of help. The problem with TC++PL in this context is that it is written for people who are already experienced programmers. I feel that it is unnecessarily hard to learn C++ from TC++PL unless you are already somewhat experienced. PPP is a better choice for most beginners and junior programmers. The key point about a good textbook is that it combines explanations of principles with carefully chosen examples. My aim is to illustrate real-world problems, techniques, and uses of language facilities with the simplest possible examples. Examples complicated by lots of details are confusing and examples without rationale can be seriously misleading. Explanations of concepts and principles without concrete examples are rarely comprehensible and typically confusing.

Ideally, the learner is guided by a competent and experienced teacher. Unfortunately, there is always a shortage of such teachers. A good teacher of software development combines a practical knowledge of software, some problems solved using software, and ability to present issues in a simple and comprehensible manner. A good teacher is also patient and persistent. Such people are rare. At least in the US it is a problem that teachers of computer science (or software development techniques) often lack industrial experience and software developers lack teaching experience. I wrote a short paper about these problems and related ones for the CACM: "*What Should We Teach New Software Developers? Why?*" Given the chronic shortage of good teachers, textbooks suitable for self study are important. In my opinion, a good textbook should contain material that allows a student to dig deeper into topics than a teacher can do in an average course – thus, all textbooks are also for self study. They should be written with that in mind.

For learners who are not native English speakers there is an added natural language barrier. They have a choice: read the original English or rely on a translation. I had that hard choice myself when I started as a computer science student. I decided to struggle through the English versions of books (because there were more and better books in English on the topics that interested me) and eventually became comfortable with that. However, I think that books in the student's native language are the ideal for beginners – reading and writing English comes later with the need to interact with the global technical community.

When it comes to programming and C++, books in the student's native language are often translations. That makes the role of the translator very important. It is not easy to write a good translation of a technical book and a book for novices needs to be both accurate and colloquial. The translator needs to be both technically savvy and a good writer. As with teachers, this is a relatively rare combination. In some countries, translations are now done by one or two individuals (more translators would lead to inconsistent style and terminology) working closely together and supported by a larger review committee. Producing a good translation is an effort comparable to writing the book in the first place. To complicate matters, that effort must often be done promptly or the material could be quite dated by the time the translation appeared. Some students still find it worth their time and money to read both the translation and the original – if nothing else that improves the student's ability to read English and ensures that more time is spent with the text.