These are questions asked by the Japanese journalist Takashi Toyota in an interview for codezine (a Japanese online magazine). You can find the Japanese language version here:

1. ://codezine.jp/article/detail/2971
2. http://codezine.jp/article/detail/3054
3. ://codezine.jp/article/detail/3220

7/12/2008:

Question 1: This is something rather personal. These days media people introduce you to their readers as C++ inventor, C++ creator and the like. How does it sound to you? Are you willing to accept the caption C++ inventor? Is it annoying you? I have once read the following article.

> With the AT&T break up, are projects like this going to go on? A lot of things will go on. It's a difficult times for fundamental research anywhere, from the universities to software engineering. So these are rough times in general, but research will go on, and what I'm doing here has always been research with a small "r." Always trying to serve a user community well. Trying to get fundamental ideas--which I don't claim to be mine. I borrowed most of my great ideas from Simula with acknowledgment. Trying to actually engineer them and popularize them and get them into real use. There never was a C++ project. It was just me building some tools and helping some people design networks and things like that. It just grew up out of a need and out of some ideas for fulfilling that need. It was only much later that there was a project. C++ was not the result of a planned, large, budgeted project. The first commercial C++ release also was by an order of magnitude the cheapest product ever produced by Bell Labs.

(Quote from http://www-old.cs.uiuc.edu/news/alumni/win96/stroustrup.html)

> ➔ When I can, I say "the designer and original implementer of C++." That's precise and maybe a bit understated because it doesn't mention the 20 years of effort refining and evolving C++, my work in the ISO C++ standards committee, my writings, etc. However, it is still far too verbose for people who want to stick a snappy label on a book cover or under a photograph. In such contexts, there are no room for even the distinction between design and implementation (I think it was essential for C++ that I did both). So it boils down to "if you can use only one adjective, which one should it be?" Given that, "creator" and "inventor" don't seem all that bad, even though I think that a short paragraph would be more likely to prevent misinterpretation of my role.
>
> In the bigger picture, I hope that those overly short labels are less important than my long-term effort to document the design and evolution of C++:
>
> - Bjarne Stroustrup: a language in and for the real world: C++ 1991-2006. ACM -III. June 2007.
> - B. Stroustrup: History of C++: 1979-1991. Proc ACM History of Programming Languages conference (HOPL-2). March 1993.

- B. Stroustrup: Design and Evolution of C++. Addison Wesley, ISBN 0-201-54330-3. March 1994.

Those writings focus primarily on the design of C++, but of course they also reflect my roles in its development.

I think too little emphasis is placed on the documenting the history of programming languages and the fundamental parts of our software development culture. Software is fundamental in our modern world, yet the work on fundamentals is so poorly documented that complete ignorance is almost standard and weird myths are common. For example, I recently tried to find photographs of the designers of the major programming languages. This turned out to be unexpectedly hard: even the Computer History Museum didn't have a complete "set" of decent photos of the designers of the major languages. The history of hardware seems to be better understood and better documented.

Question 2: How is your new book going? Almost finished? I know you are less happy with C++ books currently available at the bookstore. How do you characterize your book? I have looked at the contents more than 10 times. The book begins with "The Basics". I noticed Chapter 3 has three big concepts. They are Objects, Types, and Values in this order. At a first glance, I was a bit upset with the order. Why should Objects come before Types?

➔ The book is "in production." That means that people are working on the detailed layout, the illustrations, the quality of the English, the index, etc. The book should be in print in late October. The technical contents have been complete for a while.

I don't think you need to worry about the order of topics. Here is a sentence from the beginning of Chapter 3:

> "To read something, we need somewhere to read into; that is, we need somewhere in the computer's memory to place what we read. We call such a "place" an *object*. An object is a region of memory with a *type* that specifies what kind of information can be placed in it. A named object is called a *variable*. For example, character strings are put into **string** variables and integers are put into **int** variables."

I do not start out with elaborate class hierarchies or long explanations of the importance of dynamic binding.

The basic idea for the design of the freshman programming class (and the book) was to work backwards from what is required to start a first project aimed for use by others. That list of requirements defines the ideal set of topics. Naturally, we can't completely cover all that (even assuming suitable supervision for that hypothetical next project) in a semester. You couldn't train a plumber in three months let alone an acceptable high-school violinist. To compare, learning the basics of a natural language takes upwards of three years. Yet, we succeed in assembling a toolset of concepts and techniques. Students have reported that they have put what they learned to good use on their first real projects.

Here is an explanation of the structure of the book:

This book consists of four parts and a collection of appendices:

- *Part I "The basics"* presents the fundamental concepts and techniques of programming together with the C++ language and library facilities needed to get started writing code. This includes the type system, arithmetic operations, control structures, error handling, and the design, implementation, and use of functions and user-defined types.
- *Part II "Input and output"* describes how to get numeric and text data from the keyboard and from files, and how to produce corresponding output to the screen and to files. Then, it shows how to present numeric data, text, and geometric shapes as graphical output, and how to get input into a program from a graphical user interface (GUI).
- *Part III "Data and algorithms"* focuses on the C++ standard library's containers and algorithms framework (the STL). It shows how containers (such as **vector**, **list**, and **map**) are implemented (using pointers, arrays, dynamic memory, exceptions and templates) and used. It also demonstrates the use of standard library algorithms (such as **sort**, **find**, and **inner_product**).
- *Part IV "Broadening the view"* offers a perspective on programming through a discussion of ideals and history, through examples (such as matrix computation, text manipulation, testing, and embedded systems programming), and through a brief description of the C language.
- *Appendices* provide useful information that doesn't fit into a tutorial presentation, such as surveys of C++ language and standard library facilities, and descriptions of how to get started with an integrated development environment (IDE) and a graphical user interface (GUI) library.

The order of topics is determined by programming techniques, rather than programming language features.

I characterize my approach as "depth-first." It is also "concrete-first" and "concept-based." Part I resembles a traditional course except that it moves very fast and use higher-level types, such as string and vector right from the start, rather than being constrained by the built-in types. By the end of Part I, the students can use and write simple classes. Part II is where object-oriented programming (uses of inheritance) comes into the picture and Part III is heavily influenced by generic programming. Part III is also where pointers and free store enter the picture.

8/15/2008:

Question 1: Is your new book really an introductory version?
Thank you for introducing your new book. It's clear that many developers here are very interested in it. You characterize your new approach as "depth-first", "concrete-first and "concept-based". To be frank, all those characters do not seem to be easy to understand. Would you give us some more details about them? Your "preface" says that you have a fundamental assumption. It's about "professionalism". In this context, your book is written for serious readers, as C++ was designed for

the use of serious professional programmers. I know you have been repeating over the past several years that real-world applications should not be software toys. I also understand that expressing real-world ideas is a very serious task. Is your new book really an introductory version? Who are your ideal readers?

➔ I think readers who will benefit most from my new book will be individuals from one of three groups:

1. Someone who has never programmed before but is willing to work hard to get a good start with programming. Maybe such a person would like to become a programmer or find some other role in the software industry.
2. Someone who have been taught programming but feel the need to write better code and wants to understand why some techniques leads to more correct and maintainable code than others. I feel that a lot of teaching in programming is very weak in its treatment of principles and their practical application, so my book fills a need.
3. Someone who have programmed in a language that is not C++ and would like to become a good C++ programmer, rather than just carry over habits and techniques from another language. Different languages have different strengths and weaknesses and different styles are idiomatic in different languages(often for good reasons), so blindly carrying over habits (say from a scripting language to C++) is nowhere near optimal.

All three groups are novices, but novices in different ways. They all need a tutorial and it remains to be seen to which extent my book can serve all three. My priorities when writing the book was to focus on [1], but never forget [2] and [3]. My guess is that [1] will typically have a teacher to help them, whereas [2] and [3] will more often be self study.

What would be the *ideal* reader? Obviously, intelligence, patience, willingness to work hard, and a broad base of knowledge would help. Those are the attributes of a good student in any field. In most aspects programming is “just another high level skill,” but importantly it belongs to the skills that require both understanding of fundamentals and practical application. Programming is not *just* theory and it is not *just* a bunch of simple techniques. I think of learning programming as similar to learning to play an instrument and to acquiring a new foreign language. In all cases, it is easy to learn the task badly and get stuck at a low level of competence. On the other hand, a judicious combination of principles and practice can set a person off on a long, pleasurable, and profitable path of ever-increasing mastery. A willingness to try new things and not to be dismayed when they don’t all work out is also most important. Like other complex skills, you have to work along with incomplete skills for a long time. You cannot wait until you “know all” and then start to write code.

You are right that "depth-first", "concrete-first” and "concept-based” are hard ideas to grasp in the abstract. To really understand these concepts (any concept), you need both an explanation of the general idea (concept) and an example. If you are given just a description of the abstract idea, you have a hard time figuring out if you are using it as intended. Conversely, if you are given just an example, you have a hard time generalizing what you see to a wider range of uses.

So by “concrete-first” I mean that give concrete examples before explanations of the general principles involved. For example, I’ll show uses of function calls before explaining about argument passing and I use **vector<string**> long before I explain how to build **vector** and **string**. I am pretty sure that “concrete first” is the way most people most easily learn in essentially all areas. The alternative is often to present pages of “theory” before getting to an example or to completely explain a set of language facilities before showing a realistic of them.

By “concept-based” I mean that the aim and focus of the book is to get the student to understand the concepts/ideas and techniques that underlie good software (By “good” I mean correct, comprehensible, maintainable, etc.). Students must understand why something works or they will invent (typically erroneous) reasons and (mis)apply those “lessons.” Also, they will cut&paste rather than building new abstractions because the lack of understanding will lead to excess imitation and fear of novelty. “How to design interfaces”, “how to use invariants”, and “when to use inheritance” are examples of concepts and their applications. Programming is more than stringing **if**-statements together.

By “depth-first” I mean that I “rush” through the early chapters without explaining every detail (to get deep into the language). The aim is to collect a collection of language features and techniques to be able to present semi-realistic examples as soon as possible. If the student has to understand every detail of every language feature used (I think of that as “breath”) we are forced into a bottom-up approach where we never (in a semester course) get to anything interesting or useful. For example, I do not explain all or C++’s built-in types and their interactions, I just show uses of **bool**, **char**, **int**, and **double** and postpone more detailed explanations until much later; **vector** and **string** are used for the very start.

Why do I emphasize “professionalism” in a beginner’s book, and what do I mean by that? My view of programming is as a tool to get results; it can, of course, also be seen as a beautiful intellectual activity of its own or as an exciting hobby, but that’s secondary to me. I aim to teach people who go on to build systems on which my life and livelihood will depend one day. I want them to be responsible and competent. I want them to be economical with their time and energy. I want them to build the best systems they can. Obviously, when I designed the book and the course that it supports, I was influenced by the fact that I was (am) teaching engineers, some of whom may in a few years time be programming the plane in which I fly or some medical gadget on which I may then depend. On the other hand, I teach engineers *because* I want the people who build such systems to be competent. By using the word “professionalism,” I mean to emphasize a responsible attitude to system development and (at least) a minimal competence. I am no fan of course focused on examples featuring class hierarchies of cute animals and great mind-bending puzzles (say, Towers of Hanoi). I’d rather see examples showing people to how to validate their inputs and how to structure their code to make undetected errors less likely. I *like* clever puzzles, but prefer to do them in my spare time.

It can be a problem that not all organizations appreciate professionalism and that software development does not have professional body that defines standards for the field (as have engineers and medical doctors). Professionalism is necessary not just among developers, but

also among their managers. As I pointed out in TC++PL, quality software development puts major demands on technical managers. Unless a manager can say "this product is not ready, we cannot ship," we can't expect a developer working for him (or her) to do so. Professionalism cannot be just individual ethics; it must be backed up by standards. Unfortunately, I do not know who I would trust to set such standards. Software development is still a very young field compared to, say, medicine or civil engineering.

Question 2: Why is C++ community still growing?
Many people tend to say that learning C++ is not easy. However, recent news coverage indicates that C++ community is still growing. How do you explain this? C++ allows us to manipulate hardware directly. Do you think this C++ strength attract more and more developers from dynamic language world? These days I hear people talk more about embedded systems programming than about Web applications. Is it true C++ is increasing its number of users?

➔ The C++ community is growing in some areas and shrinking in others. It is extremely hard to find hard numbers, but my impression (based on email, conversations, conferences, books, endless visits to industrial users, sizes of vendor support groups, etc.) is that the total number of C++ users are probably is ever so slightly on the increase. I see "proofs" that C++ is declining, but those are typically exclusively based on relative measures. C++ is not growing proportionally with the number of simple web applications and the number of web sites. After all, C++'s main strengths are in infrastructure and resource constrained applications and for such areas I see no signs of shrinkage. I do believe that it is growing in absolute terms and the "portfolio" of major C++ application is not shrinking. For example, see Vincet Lextrait's documentation of historical trends for language use in major systems: http://www.lextrait.com/vincent/implementations.html (Dr. Lextrait is an executive in the French software industry, responsible for much of the software for the Amadeus airline reservation system).

But why isn't C++ disappearing as many have fantasized about since its earliest years? It is now more than ten years since prominent people from Sun declared that "Java will kill C++ completely within two years!" Since then, the C++ community has more than trebled in size and Sun has chosen C++ for some of their key applications, such as their JVM and OpenOffice. Fortunately, there is room for many languages in the world and the world is better for having this diversity. There is no language that is perfect for everything and everybody. C++ has fundamental strengths – even compared to more recent languages:

- it has a machine model that directly reflect real hardware (so it runs everywhere),
- it provides abstraction mechanisms that can be used to provide a wide variety of useful mechanisms at minimal cost (so it can be used for just about anything),
- it has a formal standard process (so that older code is not broken accidentally or for commercial gain), and
- it has a large and lively user community that keeps throwing up interesting new ideas.

Probably my HOPL-iii paper: Evolving a language in and for the real world: C++ 1991-2006 (http://www.research.att.com/~bs/hopl-almost-final.pdf) is the best discussion of those

strengths, the weaknesses of C++ (every language has some), and how the implications played out.

I do not know if C++ attracts users from the Web world. It should attract some as they have to deal with issues of scale (and some of the most prominent Web applications are written using C++), but my impression is that many don't have the background to appreciate the sources of C++'s success or to understand why it differs from a scripting language in pretty fundamental ways. Conversely, C++ should loose programmers to the web applications languages because some applications that used to need C++ now can be written in less general languages. In those cases, C++ developers have been known to seek out new systems programming tasks, rather than o stay with the application. I think that – given a choice – people tend to seek out languages and tasks that suit their talents.

10/18/2008

Question : What does the standard committee do for C++ novices?

As always, I had to spend a lot of time reading many pages before compiling this question. From your paper, I came to know that about 90 out of 100 initial suggestions for C++0x were related to language features. I was disappointed with that! Those numbers seem to tell us that the C++ evolution is lead by language experts. Compiler writers and vendors show much interest in language details but C++ novices tend to learn libraries and start writing applications based on them. C++ students are not library builder.

➔ There are a lot of compiler, tools, and language experts on the committee. Those and library builders dominate, and that's a pity. Ideally, we'd have a majority of experienced developers to help guide the work. However, the committee is not alone in its fault here. Essentially every one of those 100 proposals and maybe 200 more distinct "mere suggestions" came from people outside the committee. A member of the committee rarely just comes up with a proposal on his own. The community is the source of much of the pressure to add language features; just watch any web discussion on how to improve the language and you'll find a chorus of demands for new features together with some (often loud and unconstructive) complaints that the language is too large already.

Fortunately, most of the about 50 proposals that were accepted are so minor that most users will say something like "Huh? We couldn't do that before?" Examples are creating an alias for a template with a template argument bound and using a local type as a template argument:

```
// define Vec<T> to be an alias for vector<T,My_allocator<V>>:
using template<class T> Vec = std::vector<T,My_allocator<T>>;

void f()
```

```
{
        enum Color { blue, green, red };
        std::vector<Color> vc;        // not allowed in C++98
        // …
}
```

The alias mechanism was frequently requested under the name “template typedef.” Did you know you couldn’t do that? Did you know that the definition of **vc** in **f()** was not allowed and why? Well, both work in C++0x, and you don’t even have to add a space between the two >s in **vector<T,My_allocator<V>>**.

So, one thing that C++0x does for novices is saving them from unnecessary restrictions in a number of places. To do so was a stated aim: “To make the language rules more general and uniform.” That’s one of the two ways of making a language easier to use for novices. The other is “provide features that directly support doing simple things simply.” Most of those would be new libraries, and there are preciously few of those.

I would have liked to see many more libraries, but we did get a few: regular expressions, hashed containers (e.g. **unordered_map**), smart pointers, tuples, threads, time duration, an asynchronous return mechanism (**future**). Some of these will help novices as well as experts. None of these are new – you could have used them (or close equivalents) for years, but they now become standard and shipped with every implementation. That especially helps novices, who are often unwilling to download and install a key component (e.g. libraries from boost.org), and people in companies with policies against the use of non-standard libraries they didn’t write in-house.

Much of the help for novices comes from new features that makes the standard library easier to use and faster. My favorite is the initializer lists:

```
vector<string> names = { "Aho", "Weinberger", "Kernighan" };
map<const char*,string> designers =
        { {"C", “Ritchie"}, { "C++", "Stroustrup" }, { "BCPL", "Richards"} };
```

None of the C++98 alternatives are as simple and direct.

Question: I found two one-line sample codes in the paper. They are quite short but interest me a lot. The codes are as follows;

```
auto q = find(vi, 7);

auto q = find(vi.begin(), vi.end(), 7);
```

Are those codes are interchangeable in C++0x? If so, which code would you recommend in the classroom? To use the first code, I think we have to learn a lot about C++ history and new language extensions such as type deduction and interface contract. The second code looks wordy but is easier to understand. At least to me. To find something, we first define its search range. Am I wrong? To

meet Dr. Stroustrup, we search U.S for you first, not Japan. Any comment would be appreciated here. Do you think the first code makes compiler writers happy, not C++ learners?

➔ Actually, the container forms of the standard algorithms are not provided in the standard, so if you want them, you have to define them yourself. For example:

```
template<Container C, class V>
    C::const_iterator find(const C& c, const V& v)
        requires HasEqualTo<C::value_type,V>
{
        return find(c.begin(),c.end());
}
```

This is basically a formal notation of what you'll have to understand to use **find()** on a **vector**. Fortunately, it is easier to use an operation than to specify/define/implement it.

I would have liked to see such container forms of algorithms in the standard. However, how do we know that one piece of code is better than another? I prefer the container form

```
auto q = find(vi, 7);
```

while others consider the iterator form clearer

```
auto q = find(vi.begin(), vi.end(), 7);
```

Is this just aesthetics? First note that both give the same answer and given decent inlining (as provided by current compilers) there is no performance difference. This is purely a question of programming style. The way I would phrase the fundamental question is "Which use of **find()** expresses our intent most clearly?" Let's add comments:

```
auto q = find(vi, 7);   // find 7 in vi
                        // i.e., find the first 7 in vi
                        // i.e., find the first 7 in [vi.begin():vi.end())

auto q = find(vi.begin(), vi.end(), 7); // find the first 7 in [vi.begin():vi.end())
```

That is, the precise statement of the meaning of first call is almost exactly the second call. But how precise do we want to be? I think that "find **7** in **vi**" is the way most people think of that operation. Thinking about a container as a sequence, being explicit about "first", and "first" as "the first element in that sequence" secondary in their minds. For that reason, I'd prefer **find(vi, 7)**. Secondarily, I'll observe that that's less to type and opens fewer opportunities for mistakes. For example:

```
auto q1 = find(vi.begin(), vi.begin(), 7);      // oops!
auto q2 = find(vi.begin(), vj.end(), 7);        // oops!
```

However, every notation can of course be misused and we absolutely need the sequence form because it is the more fundamental and general. For example, given only the container form, it would be complicated to find something in half a container; with the sequence form it's easy:

```
auto middle = vi.begin()+vi.size()/2;
auto q1 = find(vi.begin(), middle, 7);
auto q2 = find(middle, vi.end(), 7);
```

Question : Which paper(s) do you want us to read?

To compile this simple question, I visited many many Web sites such as "http://www.open-std.org/jtc1/sc22/wg21/", the official site and ://www.research.att.com/~bs/WG21.html, your summary site. All I could understand is that all those papers are written for the committee members, not for ordinary people like me. How do you educate newcomers? I suspect C++ experts do not speak the same language as novices. Do you expect them to produce good tutorial books? Not many C++ books satisfy you, the original C++ designer. There are many developers in the world who want to get a big picture about C++0x. To do that, would you list two or three must-read papers for us?

➔ That's right. The WG21 papers are written by and for committee members who are experts in some aspect or other of C++. Even the tutorial parts of these papers (when there are any) are hard because they are often tutorials to help experts to understand the most obscure details of a proposal. The draft standard itself is probably the worst introduction even though it is a comprehensive high-quality document: it contains essentially no introductory material, has relatively few examples, has endless details, and is written in a complicated semi-formal language.

Obviously, there will have to be an effort to educate not just novices but also the many current C++ programmers and people who once knew C++ and will have another look at it when C++0x becomes widely available. I'll do my part – with papers, interviews, talks, and eventually books – but many people will be contribute in a wide variety of ways. Quite a few people in the committee are accomplished in tutorial presentations and many have friends who are.

One of C++'s problems have always been that its popularity outstripped the availability of quality tutorial material so that some people were confused and others suffered from material that (IMO) misrepresented what C++ was and even what it was supposed to be.

My new book ("Programming: Principles and Practice Using C++") uses C++98; after all, that's what available. I would have liked to use C++0x because that will be easier to use for teaching programming. So, as soon as there are industrial strength implementations available, I'll produce a new edition of that book. Until then, we can try C++0x features on experimental implementations and the major implementers will soon start providing C++0x features and libraries – in fact, this is already happening. Initially, I suspect we will see explanations of individual features to be published. I suspect I might write a few such

"feature tutorials" myself, but the difficult job will be to present C++0x as a whole rather as a set of disjoint features.

Just now, there is not much introductory reading material available:

- My interviews answer some questions (http://www.research.att.com/~bs/interviews.html),
- The last third of my HOPL3 paper addresses the overall design of C++0x (Bjarne Stroustrup: a language in and for the real world: C++ 1991-2006. ACM -III. June 2007. (incl. slides and videos) http://www.research.att.com/~bs/hopl-almost-final.pdf),
- There is a solid, but rather academic, paper on concepts (Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine: : Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006. (http://www.research.att.com/~bs/oopsla06.pdf))
- My extended foreword to Japanese translation of "The Design and Evolution of C++". January 2005 is not completely outdated (http://www.research.att.com/~bs/DnE2005.pdf).

I hope that a year from now I'll be able to give a far more constructive answer to this question.

Question; Is C++0x still a valid term?

The standardization work is still in progress.

➔ Until the ink is dry on the standard, I think C++0x is the right term. That's what it has been called for years and that may still be what it'll be called in the future. It is currently a *draft* standard and will be improved over the next year. The committee is not accepting new proposals, only suggested corrections. This time next year, we hope that the final technical vote will be done (making it a final draft standard). After that, there is a delay while the ISO and other standards bureaucracies add their formal approval, but unless something unexpected happens, we'll know the standard to the last comma in 2009. The question will then be whether we call it C++09 or C++10. Until then, C++0x it is!