"Any clod can have the facts, but having opinions is an art."
Charles McCabe, *San Francisco Chronicle*

The *Real* Stroustrup Interview

The father of C++ explains why Standard C++ isn't just an object-oriented language.

In Design and Evolution of C++ *(Addison Wesley, 1994), Bjarne Stroustrup argued that "a programming language is really a very tiny part of the world, and as such, it ought not be taken too seriously. Keep a sense of proportion and most importantly keep a sense of humor. Among major programming languages, C++ is the richest source of puns and jokes. That is no accident."*

For the past few months, a hoax interview between Stroustrup and Computer *has been making the rounds in cyberspace. While we regret the incident, it offers us a welcome opportunity to have the father of C++ share his insights on Standard C++ and software development in general. We can also attest to his continued sense of proportion and humor—he suggests that the fictitious interview would have been a much funnier parody had he written it himself.*

—Scott Hamilton, Computer

STANDARD C++

Computer: ISO approved the Standard C++ in November 1997, and you published a third edition of your *The C++ Programming Language* (Addison Wesley, 1997). How has C++ evolved over the past few years and what does the ISO standard mean for the C++ community?

Stroustrup: It is great to finally have a complete, detailed, and stable definition of C++. This will be a great help to the C++ community in myriad direct and not-so-direct ways. Obviously, we will get better implementations as compiler providers start shifting attention from catching up with the standards committee to quality-of-implementation issues. This is already happening.

Standards-conforming implementations will prove a boon to tools and library suppliers by providing a larger common platform to build for.

The standard gives the programmer an opportunity to be more adventurous with new techniques. Programming styles that used to be unrealistic in production code are becoming realistic propositions. Thus, more flexible, general, faster, and more maintainable code can be written.

Naturally, we should keep cool and not indulge in orgies of "advanced" techniques. There are still no miracles, and the best code is still the code that most directly matches a sound design. However, now is the time to experiment and see which techniques will suit particular people, organizations, and projects. Much of *The C++ Programming Language* is devoted to these techniques and the trade-offs they represent.

The most visible aspects of what makes this progress feasible are the "new" major language facilities—templates, exceptions, runtime type information, and namespaces—and the new standard library. The minor improvements to language details are also important. We have had most of the facilities available for years now, but haven't been able to rely on them where we needed portability and production quality. This has forced us to take suboptimal approaches to design and implementation of our libraries and applications. However, soon (this year) all major implementations will provide solid support for Standard C++.

From a language and programming-techniques perspective, the most significant change to C++ is the continuing increase in emphasis on statically verifiable type safety and the increased flexibility of templates. Flexible templates are necessary to make the emphasis on type safety pay off in terms of elegance and efficiency.

Feasibility of new techniques

Computer: You say that "often what an experienced C++ programmer has failed to notice over the years is not the introduction of new features as such, but rather the changes in relationships between features that make fundamental new programming techniques feasible." Could you give some examples of how this might be applicable to Standard C++?

Stroustrup: It is easy to study the rules of overloading and of templates without noticing that together they are one of the keys to elegant and efficient type-safe containers. This connection becomes obvious only when you look into the fundamental algorithms that are common to most uses of containers. Consider the program segment in Figure 1a. The first call of `count()` counts the number of occurrences of the value 7 in a vector of integers. The second call of `count()` counts the number of occurrences of the

Editor: Will Tracz, Lockheed Martin Federal Systems, MD 0210, Owego, NY 13827-3998; will.tracz@lmco.com

value "seven" in a list of strings.

There are ways of achieving this in every language that allows overloading. Given templates, we can write a single function that handles both calls, as shown in Figure 1b. This template will expand into optimal code for the two calls. This `count()` is roughly equivalent to the standard-library `count()` so a user will not have to write it. The `count()` function itself relies on overloading: Obviously, the equality operator == is overloaded for integers and strings (with the obvious meaning). In addition, * and ++ are overloaded to mean "dereference" and "refer to next element" for the iterator types for vectors and lists; that is, * and ++ are given the meaning they have for pointers.

This definition of `count()` illustrates a couple of the key techniques used in the containers and algorithms part of the C++ standard library. These techniques are useful, and they are not trivially discovered given only the basic language rules.

Consider a slightly more advanced variation of the `count()` example shown in Figure 2a. Instead of counting occurrences of a value, `count_if()` counts the number of elements that fulfill a predicate. For example, we can count the number of elements with a value less than 7 as shown in Figure 2b.

Generalizing this technique to handle comparisons with any value of any type requires us to define a function object—that is, a class whose objects can be invoked like functions, as shown in Figure 3a. The constructor stores a reference to the value we want to compare against, as in Figure 3b, and the application operator (`operator()`) does the comparison. In other words, count the number of elements in `vi` that have a value less than the integer 7, and count the number of elements in `ls` that have a value less than the string "seven."

It may be worth mentioning that the code generated is very efficient and in particular does not use function calls—or similar relatively slow operations—to implement comparisons, fetching the next element, and so forth.

Undoubtedly, this code will look strange to someone who is not a C++ programmer and even to C++ programmers who have not yet internalized the latest techniques for using templates and overloading—and that, of course, is part of the purpose of presenting the examples here. However, the bottom line is that these techniques allow you to write efficient, type-safe, and generic code. People familiar with functional languages will note that these techniques are similar to techniques pioneered in those languages.

```
void f(vector<int>& vi, list<string>& ls)
{
 int n1 = count(vi.begin(),vi.end(),7);
 int n2 = count(ls.begin(),ls.end(),"seven");
 // ...
}
```
(a)

```
template<class Iter, class T>
int count(Iter first, Iter last, T value)
{
 int c = 0;
 while (first!=last) {
   if (*first == value) c++;
   ++first;
 }
 return c;
}
```
(b)

Figure 1. Changes in relationships between features can make fundamental new programming techniques feasible: (a) A count algorithm for counting the number of occurrences of the value 7 in a vector of integers, as well as the number of occurrences of the value "seven" in a list of strings becomes (b) a single function that handles both calls using a template.

```
template<class Iter, class Predicate>
int count_if(Iter first, Iter last, Predicate pred)
{
 int c = 0;
 while (first!=last) {
   if (pred(*first)) c++;
   ++first;
 }
 return c;
}
```
(a)

```
bool less_than_7(int i) { return i<7; }

void f2(vector<int>& vi, list<int>& li)
{
 int n1 = count_if(vi.begin(),vi.end(),less_than_7);
 int n2 = count_if(li.begin(),li.end(),less_than_7);
 // ...
}
```
(b)

Figure 2. More advanced variation of the count example that (a) counts the number of elements that meet a predicate—for example, (b) the number of elements with a value less than 7.

Language features are fundamentally boring in isolation and they can distract from good system building—only in the context of programming techniques do they come alive.

The Standard Library

Computer: How was the Standard Library defined and what impact will it

```
template <class T> class Less_than {
 const T v;
public:
 Less_than(const T& vv) :v(vv) {} // constructor
 bool operator()(const T& e) { return e<v; }
};
```

(a)

```
void f3(vector<int>& vi, list<string>& ls)
{
 int n1 = count_if(vi.begin(),vi.end(),
   Less_than<int>(7));
 int n2 = count_if(ls.begin(),ls.end(),
   Less_than<string>("seven"));
 // ...
}
```

(b)

Figure 3. To handle comparisons with any value of any type, we define (a) a class whose objects can be invoked like functions (function object), with the constructor storing (b) a reference to the value we want to compare against.

have on the C++ community?

Stroustrup: The most novel and interesting part of the standard library is the general and extensible framework for containers and algorithms. It is often called the STL and is primarily the work of Alex Stepanov (then at Hewlett-Packard Labs, now at Silicon Graphics).

Alex worked on providing uncompromisingly general and uncompromisingly efficient fundamental algorithms which, for example, find an element in a data structure, sort a container, or count the occurrences of a value in a data structure. Such algorithms are fundamental to much computing.

Alex worked with a variety of languages and had several collaborators over the years—notably David Musser (Rensselaer Polytechnic Institute), Meng Lee (HP Labs), and Andrew Koenig (AT&T Labs). My contribution to the STL was small, but I think important: I designed an adapter called `mem_fun()`. It allows standard algorithms to be used for containers of polymorphic objects, thus tying object-oriented programming techniques neatly into the generic programming framework of the standard library.

Somewhat to my surprise, the STL matched the set of criteria for a good set of standard containers for C++ that I had developed over the years. After a year or so of discussing and polishing the STL, the ISO C++ committee accepted the STL as a key part of the standard C++ library. My criteria included

- uncompromising efficiency of simple basic operations (for example, array subscripting should not incur the cost of a function call);
- type-safety (an object from a container should be usable without explicit or implicit type conversion);
- generality (the library should provide several containers, such as vectors, lists, and maps); and
- extensibility (if the standard library didn't provide some container that I needed, I should be able to create one and use it just as I would a standard container).

By adopting STL with only minor modifications and additions, we avoided the dreaded design by committee.

Clearly, a group of part-time volunteers (the C++ standards committee) can't provide every library facility that a programmer would find useful. Consequently, a key issue was what to include in the standard library and what to leave to the industry and individuals to provide.

We decided to use "what is needed for the communication between separately developed libraries" as a guide to what to include. Thus, I/O, strings, and containers became the major focus. We included the C standard library and some facilities for numeric computation for historical reasons, but left out lots of potentially useful facilities—such as better facilities for dates and currency, regular expression matching, and graphics. Fortunately, these facilities are available commercially and/or in the public domain.

The standard library saves programmers from having to reinvent the wheel. In particular, standard containers allow both novices and experts to program at a higher level. Consider the simple program in Figure 4, which performs a simple task simply. It does so without macros, without explicit memory management or other low-level language facilities, and without resource limitations. In particular, if some joker presents the program with a 30,000 character "first name" the program will faithfully print it back at him without overflow or other errors.

Using the standard library can and should revolutionize the way C++ is taught. It is now possible to learn C++ as a higher level language. Students deal with the facilities for low-level programming only when needed and only after they've mastered enough of the language to provide a suitable context for discussing low-level facilities.

Thus, the standard library will serve as both a tool and as a teacher.

COMPLEXITY, C++, AND OOP

Computer: In *The C++ Programming Language*, you say "Ordinary practical programmers have achieved significant improvements in productivity, maintainability, flexibility, and quality in projects of just about any kind and scale." Yet some critics maintain that C++, and OO in general, are overly complex and give rise to corrective-maintenance and maintenance problems in large systems. Do these criticisms in any way correspond to your own experiences as a developer of large systems in C++?

Stroustrup: In my personal experience, OO design and OO programming lead to better code than you get from more traditional procedural approaches—code that is more flexible, more extensible, and more maintainable without imposing significant performance overheads. There is not as much hard evidence—as opposed to personal observations—as I would like, but several studies within AT&T and elsewhere support this opinion.

Two factors confound the issue: There is no general agreement on what "object-

```
int main()
{
 cout << "Please enter your first name:\n";
 string name;
 cin >> name; // read into name cout
 << "Hello" << name << "!\n";
}
```

Figure 4. Simple "Hello" program using Standard Library features to perform its task simply.

oriented" really is, and discussions rarely account for experience sufficiently. Much of "the trouble with OO" comes from people with no significant OO experience approaching an ambitious project with some partially understood—yet dogmatic and typically limited—notion of what OO code must look like.

So what is OO? Certainly not every good program is object-oriented, and not every object-oriented program is good. If this were so, "object-oriented" would simply be a synonym for "good," and the concept would be a vacuous buzzword of little help when you need to make practical decisions. I tend to equate OOP with heavy use of class hierarchies and virtual functions (called *methods* in some languages). This definition is historically accurate because class hierarchies and virtual functions together with their accompanying design philosophy were what distinguished Simula from the other languages of its time. In fact, it is this aspect of Simula's legacy that Smalltalk has most heavily emphasized.

Defining OO as based on the use of class hierarchies and virtual functions is also practical in that it provides some guidance as to where OO is likely to be successful. You look for concepts that have a hierarchical ordering, for variants of a concept that can share an implementation, and for objects that can be manipulated through a common interface without being of exactly the same type. Given a few examples and a bit of experience, this can be the basis for a very powerful approach to design.

However, not every concept naturally and usefully fits into a hierarchy, not every relationship among concepts is hierarchical, and not every problem is best approached with a primary focus on objects. For example, some problems really are primarily algorithmic. Consequently, a general-purpose programming language should support a variety of ways of thinking and a variety of programming styles. This variety results from the diversity of problems to be solved and the many ways of solving them. C++ supports a variety of programming styles and is therefore more appropriately called a *multiparadigm*, rather than an object-oriented, language (assuming you need a fancy label).

Examples of designs that meet most of the criteria for "goodness" (easy to understand, flexible, efficient) are a recursive descent parser, which is traditional procedural code. Another example is the STL, which is a generic library of containers and algorithms depending crucially on both traditional procedural code and on parametric polymorphism.

I find languages that support just one programming paradigm constraining. They buy their simplicity (whether real or imagined) by putting programmers into an intellectual straitjacket or by pushing complexity from the language into the applications. This is appropriate for special-purpose languages, but not for general-purpose languages.

I have often characterized C++ as a general-purpose programming language with a bias toward systems programming. Think of it as "a better C" that supports

- data abstraction,
- object-oriented programming, and
- generic programming.

Naturally, this support for more than one approach to programming causes more complexity than supporting only one approach. I have noticed that this view—that there are design and programming approaches with domains in which they are the best—offends some people. Clearly, I reject the view that there is *one* way that is right for everyone and for every problem. People who passionately want to believe that the world is basically simple react to this with a fury that goes beyond what I consider appropriate for discussing a programming language. After all, a programming language is just one tool among many that we use to construct our systems.

The ideal way of resolving the two views would be to have a language that provides a set of simple primitives from which all good programming styles can be efficiently supported. This has been repeatedly tried but not—in my opinion—achieved.

JAVA AND C++

Computer: That same argument for simplicity could conceivably be extended to Java, which might explain the 700,000 Java programmers that Sun claims (compared to 1.5 million C++ programmers). You maintain that Java is not the language you would have designed even if C++ needn't be compatible with C. What else do you have to say after having been asked this question for the thousandth time?

Stroustrup: These days, I'm always asked about Java, and it is very hard for me to respond. If I say something negative, I sound ungracious; if I say something positive, I feed into the commercial hype that surrounds Java and the unfortunate anti-C++ propaganda that emanates from parts of the Java community.

I encourage people to consider the two languages according to their design criteria and not just in the context of commercial rivalries. I outlined the design criteria for C++ in detail in *The Design and Evolution of C++*, and Java doesn't even start to meet those criteria. That's fine as long as programmers consider Java as a programming language among others and not a panacea. After all, C++ isn't a perfect match for Java's design aims either. However, when Java is promoted as the sole programming language, its flaws and limitations become serious.

Here are a few examples where the criteria applied to developing C++ led to significant differences. Unlike Java, C++ supports

- the ability to effectively compose a program out of parts written in different languages,
- a variety of design and programming styles,
- user-defined types with efficiencies that approach built-in types,
- uniform treatment of built-in and user-defined types, and
- the ability to use generic containers without runtime overhead (for example, the STL).

I designed C++ so programmers could write code that is both elegant and efficient. For many applications, C++ is still the best choice when you don't want to compromise between elegance and efficiency.

I wonder how people count "programmers." Is a student a programmer? Is every compiler shipped counted as a programmer? I understand the number quoted for C++; it is conservative and plausible. It is a good approximation of the number of C++ implementations sold for real money last year. I wonder if Sun's Java number is as solid. Incidentally, based on the few hard numbers I have found—such as compiler sales, book sales, and C++ course attendance—I estimate that the C++ user population is growing at 10 to 20 percent a year.

And no, I'm not a Java fan. I dislike hype, I dislike marketing of programming tools to nonprogrammers, I dislike proprietary languages, and I like *lots* of programming languages. On the technical side, Java never gave me the "Wow, that's neat!" reaction that I have had with many other languages. It provides popular language features in a form that I consider limiting.

Java has borrowed much from C++, but not as much as is often claimed and not with as much taste and understanding as one could have wished for. To deliver on some of the key promises made for it, Java must grow. This evolution may compromise Java's claim of being simpler than C++, but my guess is that the effort will make Java a better language than it is today. Currently, Java seems to be accumulating language features and "standard" libraries at quite a pace. I'm looking forward to seeing what the Java version of templates will look like; as far as I know Sun hasn't yet blessed any of the dozen or so dialects.

As Java and its community matures, it will hopefully adopt a sensible live-and-let-live philosophy. This would allow Java to take its place as one language among many in the tool chest of professional programmers.

TOOLS

Computer: In what ways have systems changed over the past few years, and what still needs to be done in the C++ arena and in systems design in general? Given C++'s relative maturity at this point, how would you rate the available C++ development tools and what still needs to be done to improve?

Stroustrup: First, I'd like to see the basic tools such as compilers, debuggers, profilers, database interfaces, GUI builders, CAD tools, and so forth fully support the ISO standard. For example, I'd like to get a database query result as an STL container or an `istream` of appropriately typed objects. Tool vendors have made a good start, but have much work to do in tools that depend on compilers and other source code analyzers.

My list of basic tools is a partial answer to the question about what has changed: Over the past few years, large numbers of programmers have come to depend on elaborate tools to interface code with systems facilities. These tools are essential to relieving programmers of tedious and error-prone tasks, but come at a risk of the programmers (and their employers) becoming captives of their tool suppliers. Often, these tools are far more complex than a traditional programming language, and there are few standards.

Java has borrowed much from C++, but not as much as is often claimed and not with as much taste and understanding as one could have wished for.

I would encourage nonproprietary standards for tools and libraries. In the short term, say 10 years, many such standards will be industry standards rather than formal ISO or IEEE standards, but it is essential for the software industry's health that key interfaces be well-specified and publicly available. With the increasing importance of standards for system-level objects such as COM and CORBA, it is particularly important that the C++ bindings to those be clean, well documented, and simple to use. Unfortunately, such "standards work"—which is beneficial to everybody—is neglected because it provides short-term competitive advantages to nobody.

I'd rather not specifically "rate" the available C++ tools. They are better than they were, and most often as good or better than any tools for any other language. However, as a programmer, I am naturally impatient for more and better quality tools. Personally, I look forward to better tools for analyzing C++ source code. I also hope that C++ implementation vendors will spend a slightly larger fraction of their budgets on improving the quality and performance of their compilers rather than concentrating too heavily on novelties. Real improvements in compilers are relatively cheap compared to what is spent on a new release of a complete C++ implementation. However, I fear that unless major users start demanding conformance testing and benchmarking, vendors will spend resources on what looks better in an advertisement.

THE FUTURE

Computer: I know you've been heavily involved in the C++ standards process, but what other projects are you involved in currently and how will you continue working to evolve C++?

Stroustrup: I'm suffering from having successfully completed a very large project. The ISO C++ standard is done. The third edition of *The C++ Programming Language* is there to acquaint serious programmers with what Standard C++ has to offer and how best to use it. And *Design and Evolution* documents the design decisions that shaped C++. There is more to be done, but nothing that requires a full-time language designer. Now is the time to enjoy the flexibility and power of C++ by writing code, rather than focusing on possible changes.

Apart from that, I'm taking the opportunity to learn a few things—a couple of new languages and something about how software is used in a large business—and planning some experiments in distributed computing. ❖

Bjarne Stroustrup is head of the large-scale programming research department at AT&T; bs@research.att.com;http://www.research.att.com/~bs/