Bjarne Stroustrup

The Design of C++0x

Reinforcing C++'s proven strengths, while moving into the future

Once or twice a day, I get e-mail suggesting improvements to C++ [1, 2]. Many suggestions are good in the sense that if one became part of the language or the Standard Library, it would make life easier for a large number of programmers. The suggestions are, of course, not all unique, so my suggestion collection contains "only" a hundred or so good ideas. The count depends on how you cluster related suggestions. You can see an incomplete list of suggested language features at http://www.research.att.com/~bs/evol-issues.html, and a wish list for Standard Library suggestions at http://lafstern.org/matt/wishlist.html. My basic opinion is that the language wish list is far too long, and that the libraries wish list is far too modest.

So, imagine you were the ISO C++ Standards committee and could decree what the next C++ Standard would look like. What would you add? What would you remove? What would you change? Seriously, take a break from reading right now and make a list. If you are still happy with your list when you finish reading this article, add arguments for your ideas and e-mail me your suggestions.

As we are revising the Standard, what are we trying to achieve? We can't expect perfect agreement on this point; after all, who are "we"? The C++ community is huge—well over 3 million programmers according to IDC last year—and incredibly diverse. We use just about every computer and operation system there is, work in just about every application area there is, and in every country on earth. This implies a diversity of needs and wishes, and to me it implies that the further evolution must be guided by a set of general principles to avoid accidentally damaging a subcommunity in the quest for improvements for some other subcommunity.

Bjarne Stroustrup is the College of Engineering Professor in Computer Science at Texas A&M University. He can be contacted at http://www.research.att.com/~bs/.

In this article, I examine the main principles we use to guide the development of C++0x, the next version of the C++ Standard. Clearly, these principles owe a lot to the "Rules of Thumb" described in [3]. That is no accident. My aim is to reinforce C++'s proven strengths rather than try to use the new Standard to encourage dramatic innovation. That said, it is not my aim to leave C++ standing still—as a living language, C++ must grow and adapt. It must become a more effective tool for solving current and future problems facing the community. I switched to first person singular here to emphasize that I'm just one individual member of a committee that contains many members. A huge range of the opinions from the C++ community are represented and what I say here appears to be backed by a strong consensus. However, the principles and their application to real problems are open to much discussion, interpretation, and experimentation. That is how it should be.

So, why don't we just accept all reasonable suggestions to make everybody happy? There are too many suggestions to analyze and adequately specify. If we accepted all suggestions there would be six or seven ways of doing anything in C++, rather than just two or three. There would be far too many features to teach and most programmers would settle into a small zone of comfort eschewing most new features. It would be extremely difficult to maintain the zero-overhead principle, partly because the implementation effort would be immense, partly because the implementation of different features would interact, and partly because many suggestions don't consider performance and rely on elaborate runtime mechanisms. This, of course, presupposes that the implementer community would accept such a Standard. I wouldn't bet on that. Please remember these concerns when you want to argue for adding "just my two new features."

Design and Evolution

C++0x will be almost 100-percent compatible with the existing Standard C++. Your existing code will, with a very high probability, be C++0x conforming if it was compliant with C++98, the existing Standard. The best guarantee for that is that the members of the committee are collectively responsible for much more old code than you are. However, we care about the future, see the evolution of C++ as a necessity, and want future code to be easier to write, more elegant, easier to maintain, and possibly even better performing than is currently feasible.

We aim for compatibility but realize that there may be cases where a large advantage is worth paying for by small incompatibilities. An obvious example is a new keyword. We try to avoid incompatibilities, but the cost of an absolute "no incompatibilities" rule would be "no change" or obscure syntax for all new features. I personally strongly dislike workarounds, such as a `__XXX` keyword supplemented by a macro:

```
#define XXX __XXX
```

The extra complexity of the workaround compared to a keyword `XXX` serves users who do not want to change code containing an `XXX` at the inconvenience of the rest of the community, added complexity to learners, and the cheap amusement of people who don't wish C++ well. There are, of course, cases where someone uses `XXX` and cannot modify the code, but then there are alternatives—don't upgrade to a new compiler or use a backwards-compatibility switch.

C++'s emphasis on general features (notably classes) has been its main strength

The idea of language evolution is often contrasted to that of language design. That's a false dichotomy. Language evolution is design. It differs from supposedly "from scratch" design by being more constrained by respect for existing code and by benefiting from more direct application of experience (feedback). In return for the difficulties imposed by compatibility, we gain the advantages of relative ease of adoption by a huge community and the smooth interoperation of old and new features. By choosing evolution, we also avoid the problem of accidentally eliminating useful programming techniques that just happened to be outside our experience or focus.

Language design can become obsessed with minor notational issues. However, a language feature is most significant if it makes a new style of programming effective to its users. In the context of C++, that means that the aim of language and Standard Library changes must be to bring new programming and design techniques into mainstream (industrial and educational) use. Only features that change the way we think are really worth providing.

To sum up: The aim for C++0x is evolution constrained by a strong need for compatibility. That aim of that evolution is to provide major real-world improvements.

Generality and Specialization

Requests for specialized facilities and minor notational improvements are very common. When provided, they rarely fail to win applause. After all, if a feature is a direct solution to a problem and doesn't significantly interact with other facilities, it is easy to explain, often easy to implement, and typically has a logically minimal expression leading to very concise pieces of code. People comparing languages using lists love such features. The snag is that the problems we face are essentially infinite, so we need an infinity of such specialized features. Examples are Pascal procedure parameters and C# delegates. The alternative traditionally offered by C++ (and before that by K&R C) is to provide very general features from which good programmers can construct solutions to a wide variety of problems. Examples are pointers and classes.

C++'s emphasis on general features (notably classes) has been its main strength, and often its lack of specialized features (such as "properties" and threads) has been seen as its main weakness. Obviously, both observations can be simultaneously true. Nevertheless, we must keep the focus on general features aimed at efficient abstraction; the huge diversity of the C++ community requires that. Features specifically tailored for, say, Windows application building or embedded systems programming, would become a huge liability if they did only what they were specialized for. C++0x will not be a "Windows language" or a "web language" or even an "embedded systems language." It will be a general-purpose language that supports those applications' areas—and more—using a common set of facilities.

One important reason to favor general mechanisms over specialized solutions to specific current problems is that the general mechanisms are likely to help with yet-un-thought-of problems: They are insurance against nasty surprises in the future. I don't want a language that can express only what I specifically designed it for.

The obvious areas where C++ could be improved for greater generality is through better support for generic programming and more flexible initialization/construction mechanisms (see example section). It is also obvious that some support for concurrency is needed as many forms of concurrent, parallel, and distributed programming are becoming common. The diversity of such approaches and techniques implies that no single mechanism can adequately cover all applications. Thus, the obvious approach is to provide very simple language mechanisms supported by libraries (built generic and object-oriented techniques).

To sum up: The aim for C++0x is to supply general language mechanisms that can be used freely in combination and to deliver more specialized features as Standard Library facilities built from language features available to all.

Experts and Novices

C++ has drifted towards becoming an "expert friendly" language. In a gathering (in person or on the Web) of experts, it is hard to build a consensus (or even interest) for something that "just" helps novices. The general opinion (in such a gathering) is typically that the best we can do for novices is to help them become experts. But it takes time to become an expert and most people need to be reasonably productive during the time it takes. More interesting, many C++ novices have no wish or need to become experts in C++. If you are a physicist needing to do a few calculations a week, an expert in some business processes involving software, or a student learning to program, you want to learn only as many language facilities as you need to get your job done. You don't want to become a language expert—you want to be (or become) an expert in your own field and know *just* enough of some programming language to get your work done. When supported by suitable libraries, C++ can be used like that—it is widely used like that. However, there are traps, pitfalls, and educational approaches that make such "occasional use" of C++ unnecessarily difficult. With a modest effort, C++0x can do much better in this area.

Consider a couple of trivial examples. Have you ever written something like this:

```
vector<vector<double>> v;
```

or this:

```
int i = extract_int(s);   // s is a string, e.g. "12.37"
```

or this:

```
vector<int>::iterator p =
  find(tbl.begin(),tbl.end(),x); // tbl is a const vector<int>
```

The most direct way of addressing the problems caused by lack of type safety is to provide a range-checked Standard Library based on statically typed containers and base the teaching of C++ on that

If not, I suspect you have either been using a different style of C++ with its own problems, or written very little C++. Admitting >> as the end of two template argument lists solves the first problem. Providing a Standard Library operation of parse a numeric value in a string would save a novice the (significant) bother of discovering the (obvious to experts) definition of a function such as `extract_int()`. Allowing the type of `p` to be deduced from its initializer would solve the third problem:

```
auto p = find(tbl.begin(),tbl.end(),x);
         // tbl is a const vector<int>
         // p becomes a vector<int>::const_iterator
```

The >> and `auto` solutions have been approved in principle for C++0x. Attacking the problem of supporting "novices of all backgrounds" requires work on both the language and the Standard Library. Concerns for education will be central; see, for example, [4].

Overloading based on concepts (see example section), would allow a further simplification:

```
auto p = find(tbl,x);
     // tbl is a const vector<int>
     // p becomes a vector<int>::const_iterator
```

Features added for novices (of all levels of expertise) should not be isolated from the rest of the language, creating some sort of "novice's ghetto." Like all features, they should scale to use in major systems, interact smoothly with other features, and provide a path for learning the complete language and Standard Library.

To sum up: C++0x must support novices of all backgrounds much better than does current C++—both through less error-prone language features and through more supportive libraries.

Type Safety

The key to elegant and efficient code is type safety using a flexible and extensible type system. C++0x will not be able to make C++ completely type safe—that would require banning arrays, uninitialized pointers, unions, narrowing conversions, C-style linking, and much more. Doing so would also cause problems with hardware access as needed in many embedded-systems applications. So what can we do? Lack of type safety is the root cause of many problems with correctness and with performance. For example:

```
void get_input(char* p)
{
   char ch;
   while (cin.get(ch) && !iswhite(ch)) *p++ = ch;
   *p = 0;
}
```

That should send chills down your spine; it really is scary. Similarly, if you are concerned with performance, you should be most unhappy with this style of generic linked list:

```
struct Link {
   Link* link;
   void* data;
};
void my_clear(Link* p, int sz)      // clear data of size sz
{
   for (Link* q = p; q!=0; q = q->link) memset(q->data,0,sz);
}
```

What can we do? Basically, all we can do is to provide alternatives to the unsafe practices and rely on tools (à la lint, but with better knowledge of the C++ type system) to detect unsafe ("traditional") use. For example:

```
string s;
cin >> s;
```

This alternative completely bypasses the problems of that low-level and sloppy `get_input()`, while being easier to use and (at least potentially) as fast. Similarly:

```
template<class In> void my_stl_clear(In first, In last)
{
   while (first!=last) *first++ = 0;
}
```

This is a complete alternative to `my_clear()`. What's wrong with `my_clear()`? It is a bit longer than `my_stl_clear()`, but that's not the point when we consider performance. The problem is a lack of type information. `Link::data` is a `void*`, so it may point to any memory location. This means that the compiler must assume that the write to `q->data` (in `memset()`) may change the contents of `*(q->link)` or even `q->link` itself. That basically disables the optimizer. Had link and data had different static types, the compiler would have assumed that they weren't aliases. So, given a reasonable optimizer, `my_stl_clear()` runs several times faster than `my_clear()`. In addition, the use of `memset()` is an example of the workarounds that we must use to access data through a `void*`: We often end up using a function

or even an indirectly called function. That, too, can be very expensive [4].

The most direct way of addressing the problems caused by lack of type safety is to provide a range-checked Standard Library based on statically typed containers and base teaching of C++ on that. This will not eliminate type errors—there will always be programmers who decide to use the "hand-coded primitive" style, programmers who must (for a variety of reasons) turn off checking when they ship, and there will always be many, many areas that are not covered by the Standard Library. The last problem will be addressed by the Standard Library, setting a standard for other libraries to meet.

Not every library should be standard

A systematic use of range checking for a version of the Standard Library implies the systematic use of exceptions, but that's okay for code that doesn't involve hard real time. By now we know how to deal with exceptions (for example, see "Appendix E" of [5]).

To sum up: C++0x will not be able to close all the loopholes in the C++ type system, but it will not introduce new holes and it will provide ways of avoiding inherently unsafe facilities—primarily through the Standard Library providing (compile-time or runtime) type-safe alternatives.

Performance and Machine Model

Performance based on a simple mapping of language features to hardware has been crucial for C++'s success (as it was for C). With the notable exception of exceptions, C++ requires minimal runtime support. Support for RTTI (`dynamic_cast` and `typeid`) and free store (`new` and `delete`) needs to be included only if you directly or indirectly use those facilities. The possibility of eliminating potentially expensive features is important in many application areas, notably embedded-systems programming. Exception handling is getting quite efficient these days, but unfortunately it is not predictable enough for hard real-time programs. Where necessary, exception handling can be disabled through compiler switches.

Nothing in C++0x will change this state of affairs. C++ is and C++0x will remain directly applicable to the most performance and resource-critical applications [6]. If C++0x adds facilities that require runtime support, those features will be designed to ensure that the additional support is required only by code that actually uses them. That is, the zero-overhead principle: "what you don't use, you don't pay for" and "what you use can be implemented without overhead compared with hand coding" is still the bedrock of C++.

The C++ model of hardware is simple: Basic types map directly to entities recognized by hardware, such as bytes, words, and registers. Sequences of objects map directly into the hardware's. Operations directly reflect machine instructions and just about everything can be done without spurious allocation, indirection, or circumlocution. The challenge will be to fit concurrency into this picture.

To sum up: C++0x will follow the zero-overhead principle and preserve a machine model that allows direct and efficient use of essentially all hardware.

Language and Libraries

A language cannot support everything, but conceivably, a large set of libraries could. Not every library should be standard. The C++ community has ample room for both a large Standard Library and a libraries industry (both commercial and open source). The committee's libraries technical report [7] raises the bar by providing facilities such as regular expression matching and hash tables. The Boost community shows other examples of what might become standard. However, even a reasonable portion of that is not enough for my taste. I'd like to see the C++ Standard Library become a much more extensive platform for systems programming, including concurrency and support for geographical distribution. "Distribution" is quite possibly beyond the committee's capacity—we are, after all, a group of volunteers with day jobs and without funds to sustain development. However, we can and must dream.

"Libraries" is an area where the committee can afford to be aggressive and opportunistic. Here, the community can provide significant (and essential) support. A library differs from a language feature in the crucial respect that it is not exclusive: If you don't like a library, you can use another. Also, a library can be developed, tested, and even deployed without the involvement of compiler vendors. C++'s mechanisms for expressing libraries—even in performance-critical and resource-constrained areas—should not be underestimated. For current examples, see Boost [8]. C++0x should be even more effective for library building.

The most commonly requested new feature for C++ is a standard GUI. The technical, economical, and political odds against that happening are immense. However, so were the odds against a good container and algorithms library for C++98. Then the STL appeared. I hope for a miracle and dream of a simple and elegant standardized interface to a variety of commercial and open-source GUI facilities. This is not a reasonable dream, but (as many have pointed out) the world is not changed by reasonable people.

To sum up: Library extensions will be preferred to language extensions. Whereas we'll be cautious and skeptical about language extensions, we'll be aggressive and opportunistic when it comes to new libraries—especially for libraries that extend the range of portable support for systems programming.

The Standard and the Real World

The ISO Standard is important to the C++ community, but it is obviously just a small part of what influences C++ program development. I don't mention other languages here, but coexistence with a huge number of systems and libraries is part of C++'s nature. That influences our considerations about language features and libraries. Furthermore, most suggestions for language and library extensions start out as more or less disguised requests to "do what language X does." I guess that every good feature in any language has, at some

point in time, been suggested for C++. As professionals, most committee members are quite familiar with a wide range of languages and our experiences do guide our design decisions. For the particularly tricky issue of C++'s relationship to ISO C, see [9].

The development of C++0x must be timely and to gain acceptance from vendors, new language features must be easier to implement than the most difficult C++98 features [ISO, 1998].

As programmers—and especially as programmers of many systems—we dislike dialects. However, there always has been and there always will be dialects. Bindings to operating systems, databases, middleware, and the like are a major source of nonstandard features. Support for prestandard facilities is another. A third source is support for "minority" needs; that is, facilities that are essential to a small community, but irrelevant to a large majority of the C++ community. My suggestion is for programmers to avoid nonstandard features whenever possible. When that is not possible—as is the case in parts of most major systems—I suggest localizing the use of the nonstandard features and accessing them through interfaces written in ISO Standard C++. One of the aims of improving the general abstraction mechanisms of C++ is exactly to make such encapsulation of nonstandard code easier. The alternative is vendor lock-in.

To sum up: We can't do everything. What we do will be guided by practical software-development concerns as well as compatibility and language-design constraints.

An Example: Supporting Generic Programming

Consider this code:

```
template<class T> class vector {
   // ...
   void push_back(const T&) { /* _ */ }
   // ...
};

vector<double> v;
v.push_back(1.2);
v.push_back(2.3);
v.push_back(3.4);
```

which is basically an example of the use of templates and the STL. From extensive real-world use and many experiments, we can consider the language facilities and programming techniques involved successful and amazingly flexible; see the STL and Boost for examples. In particular, the use of templates has become standard where performance is essential. So, how can we do better for C++0x? By "better" I mean extending what can be elegantly expressed without loss of performance compared to C++98 and which remedies problems with current use. In particular, can that `vendor` example be improved? The repeated `push_back()` is verbose and ugly, the lack of specification of the element type is a weakness leading to spectacularly bad error messages, and ideally, I don't want to expose the implementation of `push_back()`. A better version of that code would read:

```
template<Value_type T> class vector {
   // ...
   void push_back(const T&);        // just  the declaration
   // ...
};

vector<double> v;
v.push_back(1.2);
v.push_back(2.3);
v.push_back(3.4);
```

The `Value_type` used to specify the element type `T` is a "concept"; it specifies what the vector assumes about `T`. Given that we can verify that `double` is a `Value_type`, we can type check the definition of `v` without seeing the definition of `push_back()`. We would need the definition of `push_back()` if we wanted to inline, but that now becomes an implementation detail. Concepts could give us separate checking of translation units without imposing type hierarchies on template arguments or performance penalties on template users.

How could we eliminate the repeated call of `push_back()`? We could allow a vector to take an initializer list as its argument. That would require the definition of a constructor taking such an initializer list. For example:

```
template<Value_type T> class vector {
   // ...
   vector(const T*, const T*);  // sequence constructor
   // ...
};

vector<double> v = { 1.2, 2.3, 3.4 };
```

The detailed design of concepts and generalized use of initializer lists are still under intense discussion. The point here is not the details, but that I consider such facilities within the realistic grasp of C++0x.

Will This Happen?

I think C++0x will clearly reflect the rules of thumb outlined here. Lack of resources (time, people, and the like) limit what we can do and obviously (given such a large and complex task), we'll make a few mistakes. Also, some "random extensions" will slip through the net and become "odd and isolated" features in the language (much as enumerations are in C and C++). However, there is reason to expect that C++0x will be a significant improvement over current C++ for essentially all of its users and for many more users still to come.

References

[1] "Standard for the C++ Programming Language," ISO/IEC 14882.

[2] The C++ Standard (ISO/IEC 14882:2002), Wiley, 2003. ISBN 0-470-84674-7.

[3] Stroustrup, B. *The Design and Evolution of C++*, Addison-Wesley, 1994. ISBN 0-201-54330-3.

[4] Stroustrup, B. "Learning Standard C++ as a New Language," *C/C++ Users Journal*, May 1999 (http://www.research.att.com/~bs/ papers.html).

[5] Stroustrup, B. *The C++ Programming Language*, Special Edition, Addison-Wesley, 2000. ISBN 0-201-70073-5.

[6] "Technical Report on C++ Performance," ISO.IEC PDTR 18015 (http://www.research.att.com/~bs/C++.html).

[7] "Technical Report on C++ Standard Library Extensions," ISO/IEC PDTR 19768.

[8] http://www.boost.org/.

[9] Stroustrup, B. "C and C++: Siblings," "C and C++: A Case for Compatibility," and "C and C++: Case Studies in Compatibility," *C/C++ Users Journal*, July/August/September, 2002 (http://www.research.att.com/~bs/papers.html). ❑