CHAPTER ONE

C++

C++ occupies an interesting space among languages: it is built on the foundation of C, incorporating object-orientation ideas from Simula; standardized by ISO; and designed with the mantras “you don’t pay for what you don’t use” and “support user-defined and built-in types equally well.” Although popularized in the 80s and 90s for OO and GUI programming, one of its greatest contributions to software is its pervasive generic programming techniques, exemplified in its Standard Template Library. Newer languages such as Java and C# have attempted to replace C++, but an upcoming revision of the C++ standard adds new and long-awaited features. Bjarne Stroustrup is the creator of the language and still one of its strongest advocates.

Design Decisions

Why did you choose to extend an existing language instead of creating a new one?

Bjarne Stroustrup: When I started—in 1979—my purpose was to help programmers build systems. It still is. To provide genuine help in solving a problem, rather than being just an academic exercise, a language must be complete for the application domain. That is, a non-research language exists to solve a problem. The problems I was addressing related to operating system design, networking, and simulation. I—and my colleagues—needed a language that could express program organization as could be done in Simula (that's what people tend to call object-oriented programming), but also write efficient low-level code, as could be done in C. No language that could do both existed in 1979, or I would have used it. I didn't particularly want to design a new programming language; I just wanted to help solve a few problems.

Given that, building on an existing language makes a lot of sense. From the base language, you get a basic syntactic and semantic structure, you get useful libraries, and you become part of a culture. Had I not built on C, I would have based C++ on some other language. Why C? I had Dennis Ritchie, Brian Kernighan, and other Unix greats just down (or across) the hall from me in Bell Labs' Computer Science Research Center, so the question may seem redundant. But it was a question I took seriously.

In particular, C's type system was informal and weakly enforced (as Dennis Ritchie said, "C is a strongly typed, weakly checked language"). The "weakly checked" part worried me and causes problems for C++ programmers to this day. Also, C wasn't the widely used language it is today. Basing C++ on C was an expression of faith in the model of computation that underlies C (the "strongly typed" part) and an expression of trust in my colleagues. The choice was made based on knowledge of most higher-level programming languages used for systems programming at the time (both as a user and as an implementer). It is worth remembering that this was a time when most work "close to the hardware" and requiring serious performance was still done in assembler. Unix was a major breakthrough in many ways, including its use of C for even the most demanding systems programming tasks.

So, I chose C's basic model of the machine over better-checked type systems. What I really wanted as the framework for programs was Simula's classes, so I mapped those into the C model of memory and computation. The result was something that was extremely expressive and flexible, yet ran at a speed that challenged assembler without a massive runtime support system.

Why did you choose to support multiple paradigms?

Bjarne: Because a combination of programming styles often leads to the best code, where "best" means code that most directly expresses the design, runs faster, is most maintainable, etc. When people challenge that statement, they usually do so by either defining their favorite programming style to include every useful construct (e.g., "generic programming is simply a form of OO") or excluding application areas (e.g., "everybody has a 1GHz, 1GB machine").

Java focuses solely on object-oriented programming. Does this make Java code more complex in some cases where C++ can instead take advantage of generic programming?

Bjarne: Well, the Java designers—and probably the Java marketers even more so—emphasized OO to the point where it became absurd. When Java first appeared, claiming purity and simplicity, I predicted that if it succeeded Java would grow significantly in size and complexity. It did.

For example, using casts to convert from `Object` when getting a value out of a container (e.g., `(Apple)c.get(i)`) is an absurd consequence of not being able to state what type the objects in the container is supposed have. It's verbose and inefficient. Now Java has generics, so it's just a bit slow. Other examples of increased language complexity (helping the programmer) are enumerations, reflection, and inner classes.

The simple fact is that complexity will emerge somewhere, if not in the language definition, then in thousands of applications and libraries. Similarly, Java's obsession with putting every algorithm (operation) into a class leads to absurdities like classes with no data consisting exclusively of static functions. There are reasons why math uses `f(x)` and `f(x,y)` rather than `x.f( )`, `x.f(y)`, and `(x,y).f( )`—the latter is an attempt to express the idea of a "truly object-oriented method" of two arguments and to avoid the inherent asymmetry of `x.f(y)`.

C++ addresses many of the logical as well as the notational problems with object orientation through a combination of data abstraction and generic programming techniques. A classical example is `vector<T>` where `T` can be any type that can be copied—including built-in types, pointers to OO hierarchies, and user-defined types, such as strings and complex numbers. This is all done without adding runtime overheads, placing restrictions on data layouts, or having special rules for standard library components. Another example that does not fit the classical single-dispatch hierarchy model of OO is an operation that requires access to two classes, such as `operator*(Matrix,Vector)`, which is not naturally a "method" of either class.

One fundamental difference between C++ and Java is the way pointers are implemented. In some ways, you could say that Java doesn't have real pointers. What differences are there between the two approaches?

Bjarne: Well, of course Java has pointers. In fact, just about everything in Java is implicitly a pointer. They just call them *references*. There are advantages to having pointers implicit as well as disadvantages. Separately, there are advantages to having true local objects (as in C++) as well as disadvantages.

C++'s choice to support stack-allocated local variables and true member variables of every type gives nice uniform semantics, supports the notion of value semantics well, gives compact layout and minimal access costs, and is the basis for C++'s support for general resource management. That's major, and Java's pervasive and implicit use of pointers (aka references) closes the door to all that.

Consider the layout tradeoff: in C++ a `vector<complex>(10)` is represented as a handle to an array of 10 complex numbers on the free store. In all, that's 25 words: 3 words for the vector, plus 20 words for the complex numbers, plus a 2-word header for the array on the free store (heap). The equivalent in Java (for a user-defined container of objects of user-defined types) would be 56 words: 1 for the reference to the container, plus 3 for the container, plus 10 for the references to the objects, plus 20 for the objects, plus 24 for the free store headers for the 12 independently allocated objects. Obviously, these numbers are approximate because the free store (heap) overhead is implementation defined in both languages. However, the conclusion is clear: by making references ubiquitous and implicit, Java may have simplified the programming model and the garbage collector implementation, but it has increased the memory overhead dramatically—and increased the memory access cost (requiring more indirect accesses) and allocation overheads proportionally.

What Java doesn't have—and good for Java for that—is C and C++'s ability to misuse pointers through pointer arithmetic. Well-written C++ doesn't suffer from that problem either: people use higher-level abstractions, such as iostreams, containers, and algorithms, rather than fiddling with pointers. Essentially all arrays and most pointers belong deep in implementations that most programmers don't have to see. Unfortunately, there is also lots of poorly written and unnecessarily low-level C++ around.

There is, however, an important place where pointers—and pointer manipulation—is a boon: the direct and efficient expression of data structures. Java's references are lacking here; for example, you can't express a `swap` operation in Java. Another example is simply the use of pointers for low-level direct access to (real) memory; for every system, some language has to do that, and often that language is C++.

The "dark side" of having pointers (and C-style arrays) is of course the potential for misuse: buffer overruns, pointers into deleted memory, uninitialized pointers, etc. However, in well-written C++ that is not a major problem. You simply don't get those problems with pointers and arrays used within abstractions (such as `vector`, `string`, `map`, etc.). Scoped resource management takes care of most needs; smart pointers and specialized handles can be used to deal with most of the rest. People whose experience is primarily C or old-style C++ find this hard to believe, but scope-based resource management is an immensely powerful tool and user-defined with suitable operations can address classical problems with less code than the old insecure hacks. For example, this is the simplest form of the classical buffer overrun and security problem:

```
char buf[MAX_BUF];
gets(buf); // Yuck!
```

Use a standard library string and the problem goes away:

```
string s;
cin >> s;    // read whitespace separated characters
```

These are obviously trivial examples, but suitable "strings" and "containers" can be crafted to meet essentially all needs, and the standard library provides a good set to start with.

What do you mean by “value semantics” and “general resource management”?

Bjarne: “Value semantics” is commonly used to refer to classes where the objects have the property that when you copy one, you get two independent copies (with the same value). For example:

```
X x1 = a;
X x2 = x1; // now x1==x2
x1 = b;    // changes x1 but not x2
           // now x1!=x2 ( provided X(a)!=X(b) )
```

This is of course what we have for usual numeric types, such as ints, doubles, complex numbers, and mathematical abstractions, such as vectors. This is a most useful notion, which C++ supports for built-in types and for any user-defined type for which we want it. This contrast to Java where built-in types such and `char` and `int` follow it, but user-defined types do not, and indeed cannot. As in Simula, all user-defined types in Java have reference semantics. In C++, a programmer can support either, as the desired semantics of a type requires. C# (incompletely) follows C++ in supporting user-defined types with value semantics.

“General resource management” refers to the popular technique of having a resource (e.g., a file handle or a lock) owned by an object. If that object is a scoped variable, the lifetime of the variable puts a maximum limit on the time the resource is held. Typically, a constructor acquires the resource and the destructor releases it. This is often called RAII (Resource Acquisition Is Initialization) and integrates beautifully with error handling using exceptions. Obviously, not every resource can be handled in this way, but many can, and for those, resource management becomes implicit and efficient.

“Close to the hardware” seems to be a guiding principle in designing C++. Is it fair to say that C++ was designed more bottom-up than many languages, which are designed top-down, in the sense that they try to provide abstractly rational constructs and force the compiler to fit these constructs to the available computing environment?

Bjarne: I think top-down and bottom-up are the wrong way to characterize those design decisions. In the context of C++ and other languages, “close to the hardware” means that the model of computation is that of the computer—sequences of objects in memory and operations as defined on objects of fixed size—rather than some mathematical abstraction. This is true for both C++ and Java, but not for functional languages. C++ differs from Java in that its underlying machine is the real machine rather than a single abstract machine.

The real problem is how to get from the human conception of problems and solutions to the machine’s limited world. You can “ignore” the human concerns and end up with machine code (or the glorified machine code that is bad C code). You can ignore the machine and come up with a beautiful abstraction that can do anything at extraordinary cost and/or lack of intellectual rigor. C++ is an attempt to give a very direct access to hardware when you need it (e.g., pointers and arrays) while providing extensive abstraction mechanisms to allow high-level ideas to be expressed (e.g., class hierarchies and templates).

That said, there has been a consistent concern for runtime and space performance throughout the development of C++ and its libraries. This pervades both the basic language facilities and the abstraction facilities in ways that are not shared by all languages.

Using the Language

How do you debug? Do you have any suggestion for C++ developers?

Bjarne: By introspection. I study the program for so long and poke at it more or less systematically for so long that I have sufficient understanding to provide an educated guess where the bug is.

Testing is something else, and so is design to minimize errors. I intensely dislike debugging and will go a long way to avoid it. If I am the designer of a piece of software, I build it around interfaces and invariants so that it is hard to get seriously bad code to compile and run incorrectly. Then, I try hard to make it testable. Testing is the systematic search for errors. It is hard to systematically test badly structured systems, so I again recommend a clean structure of the code. Testing can be automated and is repeatable in a way that debugging is not. Having flocks of pigeons randomly peck at the screen to see if they can break a GUI-based application is no way to ensure quality systems.

Advice? It is hard to give general advice because the best techniques often depend on what is feasible for a given system in a given development environment. However: identify key interfaces that can be systematically tested and write test scripts that exercise those. Automate as much as you can and run those automated tests often. And do keep regression tests and run them frequently. Make sure that every entry point into the system and every output can be systematically tested. Compose your system out of quality components: monolithic programs are unnecessarily hard to understand and test.

At what level is it necessary to improve the security of software?

Bjarne: First of all: security is a systems issue. No localized or partial remedy will by itself succeed. Remember, even if all of your code was perfect, I could probably still gain access to your stored secrets if I could steal your computer or the storage device holding your backup. Secondly, security is a cost/benefit game: perfect security is probably beyond the reach for most of us, but I can probably protect my system sufficiently that "bad guys" will consider their time better spent trying to break into someone else's system. Actually, I prefer not to keep important secrets online and leave serious security to the experts.

But what about programming languages and programming techniques? There is a dangerous tendency to assume that every line of code has to be "secure" (whatever that means), even assuming that someone with bad intentions messes with some other part of the system. This is a most dangerous notion that leaves the code littered with unsystematic tests guarding against ill-formulated imagined threats. It also makes code ugly, large, and slow. "Ugly" leaves places for bugs to hide, "large" ensures incomplete testing, and "slow" encourages the use of shortcuts and dirty tricks that are among the most fertile sources of security holes.

I think the only permanent solution to security problems is in a simple security model applied systematically by quality hardware and/or software to selected interfaces. There has to be a place behind a barrier where code can be written simply, elegantly, and efficiently without worrying about random pieces of code abusing random pieces of other code. Only then can we focus on correctness, quality, and serious performance. The idea that anyone can provide an untrusted callback, plug-in, overrider, whatever, is plain silly. We have to distinguish between code that defends against fraud, and code that simply is protected against accidents.

I do not think that you can design a programming language that is completely secure and also useful for real-world systems. Obviously, that depends on the meaning of "secure" and "system." You could possibly achieve security in a domain-specific language, but my main domain of interest is systems programming (in a very broad meaning of that term), including embedded systems programming. I do think that type safety can and will be improved over what is offered by C++, but that is only part of the problem: type safety does not equal security. People who write C++ using lots of unencapsulated arrays, casts, and unstructured new and delete operations are asking for trouble. They are stuck in an 80s style of programming. To use C++ well, you have to adopt a style that minimizes type safety violations and manage resources (including memory) in a simple and systematic way.

Would you recommend C++ for some systems where practitioners are reluctant to use it, such as system software and embedded applications?

Bjarne: Certainly, I do recommend it and not everybody is reluctant. In fact, I don't see much reluctance in those areas beyond the natural reluctance to try something new in established organizations. Rather, I see steady and significant growth in C++ use. For example, I helped write the coding guidelines for the mission-critical software for Lockheed Martin's Joint Strike Fighter. That's an "all C++ plane." You may not be particularly keen on military planes, but there is nothing particularly military about the way C++ is used and well over 100,000 copies of the JSF++ coding rules have been downloaded from my home pages in less than a year, mostly by nonmilitary embedded systems developers, as far as I can tell.

C++ has been used for embedded systems since 1984, many useful gadgets have been programmed in C++, and its use appears to be rapidly increasing. Examples are mobile phones using Symbian or Motorola, the iPods, and GPS systems. I particularly like the use of C++ on the Mars rovers: the scene analysis and autonomous driving subsystems, much of the earth-based communication systems, and the image processing.

People who are convinced that C is necessarily more efficient than C++ might like to have a look at my paper entitled "Learning Standard C++ as a New Language" [*C/C++ Users Journal*, May 1999], which describes a bit of design philosophy and shows the result of a few simple experiments. Also, the ISO C++ standards committee issued a technical report on performance that addresses a lot of issues and myths relating to the use of C++ where performance matters (you can find it online searching for "Technical Report on C++ Performance").* In particular, that report addresses embedded systems issues.

* *http://www.open-std.org/JTC1/sc22/wg21/docs/TR18015.pdf*

,ch01.25132 Page 8 Wednesday, March 11, 2009 4:08 PM

Kernels like Linux's or BSD's are still written in C. Why haven't they moved to C++? Is it something in the OO paradigm?

Bjarne: It's mostly conservatism and inertia. In addition, GCC was slow to mature. Some people in the C community seem to maintain an almost willful ignorance based on decade-old experiences. Other operating systems and much systems programming and even hard real-time and safety-critical code has been written in C++ for decades. Consider some examples: Symbian, IBM's OS/400 and K42, BeOS, and parts of Windows. In general, there is a lot of open source C++ (e.g., KDE).

You seem to equate C++ use with OO. C++ is not and was never meant to be just an object-oriented programming language. I wrote a paper entitled "Why C++ is not just an Object-Oriented Programming Language" in 1995; it is available online.* The idea was and is to support multiple programming styles ("paradigms," if you feel like using long words) and their combinations. The most relevant other paradigm in the context of high-performance and close-to-the-hardware use is generic programming (sometimes abbreviated to GP). The ISO C++ standard library is itself more heavily GP than OO through its framework for algorithms and containers (the STL). Generic programming in the typical C++ style relying heavily on templates is widely used where you need both abstraction and performance.

I have never seen a program that could be written better in C than in C++. I don't think such a program could exist. If nothing else, you can write C++ in a style close to that of C. There is nothing that requires you to go hog-wild with exceptions, class hierarchies, or templates. A good programmer uses the more advanced features where they help more directly to express ideas and do so without avoidable overheads.

Why should a programmer move his code from C to C++? What advantages would he have using C++ as a generic programming language?

Bjarne: You seem to assume that code first was written in C and that the programmer started out as a C programmer. For many—probably most—C++ programs and C++ programmers, that has not been the case for quite a while. Unfortunately, the "C first" approach lingers in many curricula, but it is no longer something to take for granted.

Someone might switch from C to C++ because they found C++'s support for the styles of programming usually done with C is better than C's. The C++ type checking is stricter (you can't forget to declare a function or its argument types) and there is type-safe notational support for many common operations, such as object creation (including initialization) and constants. I have seen people do that and be very happy with the problems they left behind. Usually, that's done in combination with the adoption of some C++ libraries that may or may not be considered object-oriented, such as the standard vector, a GUI library, or some application-specific library.

* *http://www.research.att.com/~bs/oopsla.pdf*

Just using a simple user-defined type, such as `vector`, `string`, or `complex`, does not require a paradigm shift. People can—if they so choose—use those just like the built-in types. Is someone using `std::vector` "using OO"? I would say no. Is someone using a C++ GUI without actually adding new functionality "using OO"? I'm inclined to say yes, because their use typically requires the users to understand and use inheritance.

Using C++ as "a generic-programming programming language" gives you the standard containers and algorithms right out of box (as part of the standard library). That is major leverage in many applications and a major step up in abstraction from C. Beyond that, people can start to benefit from libraries, such as Boost, and start to appreciate some of the functional programming techniques inherent in generic programming.

However, I think the question is slightly misleading. I don't want to represent C++ as "an OO language" or "a GP language"; rather, it is a language supporting:

- C-style programming
- Data abstraction
- Object-oriented programming
- Generic programming

Crucially, it supports programming styles that combines those ("multiparadigm programming" if you must) and does so with a bias toward systems programming.

OOP and Concurrency

The average complexity and size (in number of lines of code) of software seems to grow year after year. Does OOP scale well to this situation or just make things more complicated? I have the feeling that the desire to make reusable objects makes things more complicated and, in the end, it doubles the workload. First, you have to design a reusable tool. Later, when you need to make a change, you have to write something that exactly fits the gap left by the old part, and this means restrictions on the solution.

Bjarne: That's a good description of a serious problem. OO is a powerful set of techniques that can help, but to be a help, it must be used well and for problems where the techniques have something to offer. A rather serious problem for all code relying on inheritance with statically checked interfaces is that to design a good base class (an interface to many, yet unknown, classes) we require a lot of foresight and experience. How does the designer of the base class (abstract class, interface, whatever you choose to call it) know that it specifies all that is needed for all classes that will be derived from it in the future? How does the designer know that what is specified can be implemented reasonably by all classes that will be derived from it in the future? How does the designer of the base class know that what is specified will not seriously interfere with something that is needed by some classes that will be derived from it in the future?

In general, we can't know that. In an environment where we can enforce our design, people will adapt—often by writing ugly workarounds. Where no one organization is in charge, many incompatible interfaces emerge for essentially the same functionality.

Nothing can solve these problems in general, but generic programming seems to be an answer in many important cases where the OO approach fails. A noteworthy example is simply containers: we cannot express the notion of being an element well through an inheritance hierarchy, and we can't express the notion of being a container well through an inheritance hierarchy. We can, however, provide effective solutions using generic programming. The STL (as found in the C++ standard library) is an example.

Is this problem specific to C++, or does it afflict other programming languages as well?

Bjarne: The problem is common to all languages that rely on statically checked interfaces to class hierarchies. Examples are C++, Java, and C#, but not dynamically typed languages, such as Smalltalk and Python. C++ addresses that problem through generic programming, where the C++ containers and algorithms in standard library provide a good example. The key language feature here is templates, providing a late type-checking model that gives a compile time equivalent to what the dynamically typed languages do at runtime. Java's and C#'s recent addition of "generics" are attempts to follow C++'s lead here, and are often—incorrectly, I think—claimed to improve upon templates.

"Refactoring" is especially popular as an attempt to address that problem by the brute force technique of simply reorganizing the code when it has outlived its initial interface design.

If this is a problem of OO in general, how can we be sure that the advantages of OO are more valuable than the disadvantages? Maybe the problem that a good OO design is difficult to achieve is the root of all other problems.

Bjarne: The fact that there is a problem in some or even many cases doesn't change the fact that many beautiful, efficient, and maintainable systems have been written in such languages. Object-oriented design is one of the fundamental ways of designing systems and statically checked interfaces provide advantages as well as this problem.

There is no one "root of all evil" in software development. Design is hard in many ways. People tend to underestimate the intellectual and practical difficulties involved in building a significant system involving software. It is not and will not be reduced to a simple mechanical "assembly line" process. Creativity, engineering principles, and evolutionary change are needed to create a satisfactory large system.

Are there links between the OO paradigm and concurrency? Does the current pervasive need for improved concurrency change the implementation of designs or the nature of OO designs?

Bjarne: There is a very old link between object-oriented programming and concurrency. Simula 67, the programming language that first directly supported object-oriented programming, also provided a mechanism for expressing concurrent activities.

The first C++ library was a library supporting what today we would call *threads*. At Bell Labs, we ran C++ on a six-processor machine in 1988 and we were not alone in such uses. In the 90s there were at least a couple of dozen experimental C++ dialects and libraries attacking problems related to distributed and parallel programming. The current excitement about multicores isn't my first encounter with concurrency. In fact, distributed computing was my Ph.D. topic and I have followed that field ever since.

However, people who first consider concurrency, multicores, etc., often confuse themselves by simply underestimating the cost of running an activity on a different processor. The cost of starting an activity on another processor (core) and for that activity to access data in the "calling processor's" memory (either copying or accessing "remotely") can be 1,000 times (or more) higher than we are used to for a function call. Also, the error possibilities are significantly different as soon as you introduce concurrency. To effectively exploit the concurrency offered by the hardware, we need to rethink the organization of our software.

Fortunately, but confusingly, we have decades' worth of research to help us. Basically, there is so much research that it's just about impossible to determine what's real, let alone what's best. A good place to start looking would be the HOPL-III paper about Emerald. That language was the first to explore the interaction between language issues and systems issues, taking cost into account. It is also important to distinguish between data parallel programming as has been done for decades—mostly in FORTRAN—for scientific calculations, and the use of communicating units of "ordinary sequential code" (e.g., processes and threads) on many processors. I think that for broad acceptance in this brave new world of many "cores" and clusters, a programming system must support both kinds of concurrency, and probably several varieties of each. This is not at all easy, and the issues go well beyond traditional programming language issues—we will end up looking at language, systems, and applications issues in combination.

Is C++ ready for concurrency? Obviously we can create libraries to handle everything, but does the language and standard library need a serious review with concurrency in mind?

Bjarne: Almost. C++0x will be. To be ready for concurrency, a language first has to have a precisely specified memory model to allow compiler writers to take advantage of modern hardware (with deep pipelines, large caches, branch-prediction buffers, static and dynamic instruction reordering, etc.). Then, we need a few small language extensions: thread-local storage and atomic data types. Then, we can add support for concurrency as libraries. Naturally, the first new standard library will be a threads library allowing portable programming across systems such as Linux and Windows. We have of course had such libraries for many years, but not standard ones.

Threads plus some form of locking to avoid data races is just about the worst way to directly exploit concurrency, but C++ needs that to support existing applications and to maintain its role as a systems programming language on traditional operating systems. Prototypes of this library exist—based on many years of active use.

One key issue for concurrency is how you "package up" a task to be executed concurrently with other tasks. In C++, I suspect the answer will be "as a function object." The object can contain whatever data is needed and be passed around as needed. C++98 handles that well for named operations (named classes from which we instantiate function objects), and the technique is ubiquitous for parameterization in generic libraries (e.g., the STL). C++0x makes it easier to write simple "one-off" function objects by providing "lambda functions" that can be written in expression contexts (e.g., as function arguments) and generates function objects ("closures") appropriately.

The next steps are more interesting. Immediately post-C++0x, the committee plans for a technical report on libraries. This will almost certainly provide for thread pools and some form of work stealing. That is, there will be a standard mechanism for a user to request relatively small units of work ("tasks") to be done concurrently without fiddling with thread creation, cancellation, locking, etc., probably built with function objects as tasks. Also, there will be facilities for communicating between geographically remote processes through sockets, iostreams, and so on, rather like `boost::networking`.

In my opinion, much of what is interesting about concurrency will appear as multiple libraries supporting logically distinct concurrency models.

Many modern systems are componentized and spread out over a network; the age of web applications and mashups may accentuate that trend. Should a language reflect those aspects of the network?

Bjarne: There are many forms of concurrency. Some are aimed at improving the throughput or response time of a program on a single computer or cluster, some are aimed at dealing with geographical distribution, and some are below the level usually considered by programmers (pipelining, caching, etc.).

C++0*x* will provide a set of facilities and guarantees that saves programmers from the lowest-level details by providing a "contract" between machine architects and compiler writers—a "machine model." It will also provide a threads library providing a basic mapping of code to processors. On this basis, other models can be provided by libraries. I would have liked to see some simpler-to-use, higher-level concurrency models supported in the C++0*x* standard library, but that now appears unlikely. Later—hopefully, soon after C++0*x*—we will get more libraries specified in a technical report: thread pools and futures, and a library for I/O streams over wide area networks (e.g., TCP/IP). These libraries exist, but not everyone considers them well enough specified for the standard.

Years ago, I hoped that C++0*x* would address some of C++'s long-standing problems with distribution by specifying a standard form of marshalling (or serialization), but that didn't happen. So, the C++ community will have to keep addressing the higher levels of distributed computing and distributed application building through nonstandard libraries and/or frameworks (e.g., CORBA or .NET).

The very first C++ library (really the very first C with classes) library, provided a lightweight form of concurrency and over the years, hundreds of libraries and frameworks for concurrent, parallel, and distributed computing have been built in C++, but the community has not been able to agree on standards. I suspect that part of the problem is that it takes a lot of money to do something major in this field, and that the big players preferred to spend their money on their own proprietary libraries, frameworks, and languages. That has not been good for the C++ community as a whole.

Future

Will we ever see C++ 2.0?

Bjarne: That depends on what you mean by "C++ 2.0." If you mean a new language built more or less from scratch providing all of the best of C++ but none of what's bad (for some definitions of "good" and "bad"), the answer is "I don't know." I would like to see a major new language in the C++ tradition, but I don't see one on the horizon, so let me concentrate on the next ISO C++ standard, nicknamed C++0x.

It will be a "C++ 2.0" to many, because it will supply new language features and new standard libraries, but it will be almost 100% compatible with C++98. We call it C++0*x*, hoping that it'll become C++09. If we are slow—so that that *x* has to become hexadecimal—I (and others) will be quite sad and embarrassed.

C++0x will be almost 100% compatible with C++98. We have no particular desire to break your code. The most significant incompatibilities come from the use of a few new keywords, such as `static_assert`, `constexpr`, and `concept`. We have tried to minimize impact by choosing new keywords that are not heavily used. The major improvements are:

- Support for modern machine architectures and concurrency: a machine model, a thread library, thread local storage and atomic operations, and an asynchronous value return mechanism ("futures").
- Better support for generic programming: concepts (a type system for types, combinations of types, and combinations of types and integers) to give better checking of template definitions and uses, and better overloading of templates. Type deduction based on initializers (`auto`), generalized initializer lists, generalized constant expressions (`constexpr`), lambda expressions, and more.
- Many "minor" language extensions, such as static assertions, move semantics, improved enumerations, a name for the null pointer (`nullptr`), etc.
- New standard libraries for regular expression matching, hash tables (e.g., `unordered_map`), "smart" pointers, etc.

For complete details, see the website of the "C++ Standards Committee."* For an overview, see my online C++0x FAQ.†

* *http://www.open-std.org/jtc1/sc22/wg21/*

† *http://www.research.att.com/~bs/C++0xFAQ.html*

Please note that when I talk about "not breaking code," I am referring to the core language and the standard library. Old code will of course be broken if it uses nonstandard extensions from some compiler provider or antique nonstandard libraries. In my experience, when people complain about "broken code" or "instability" they are referring to proprietary features and libraries. For example, if you change operating systems and didn't use one of the portable GUI libraries, you probably have some work to do on the user interface code.

What stops you from creating a major new language?

Bjarne: Some key questions soon emerge:

- What problem would the new language solve?
- Who would it solve problems for?
- What dramatically new could be provided (compared to every existing language)?
- Could the new language be effectively deployed (in a world with many well-supported languages)?
- Would designing a new language simply be a pleasant distraction from the hard work of helping people build better real-world tools and systems?

So far, I have not been able to answer those questions to my satisfaction.

That doesn't mean that I think that C++ is the perfect language of its kind. It is not; I'm convinced that you could design a language about a tenth of the size of C++ (whichever way you measure size) providing roughly what C++ does. However, there has to be more to a new language that just doing what an existing language can, but slightly better and slightly more elegantly.

What do the lessons about the invention, further development, and adoption of your language say to people developing computer systems today and in the foreseeable future?

Bjarne: That's a big question: can we learn from history? If so, how? What kind of lessons can we learn? During the early development of C++, I articulated a set of "rules of thumb," which you can find in *The Design and Evolution of C++* [Addison-Wesley], and also discussed in my two HOPL papers. Clearly, any serious language design project needs a set of principles, and as soon as possible, these principles need to be articulated. That's actually a conclusion from the C++ experience: I didn't articulate C++'s design principles early enough and didn't get those principles understood widely enough. As a result, many people invented their own rationales for C++'s design; some of those were pretty amazing and led to much confusion. To this day, some see C++ as little more than a failed attempt to design something like Smalltalk (no, C++ was not supposed to be "like Smalltalk"; it follows the Simula model of OO), or as nothing but an attempt to remedy some flaws in C for writing C-style code (no, C++ was not supposed to be just C with a few tweaks).

The purpose of a (nonexperimental) programming language is to help build good systems. It follows that notions of system design and language design are closely related.

My definition of "good" in this context is basically "correct, maintainable, and providing acceptable resource usage." The obvious missing component is "easy to write," but for the kind of systems I think most about, that's secondary. "RAD development" is not my ideal. It can be as important to say what is not a primary aim as to state what is. For example, I have nothing against rapid development—nobody in their right mind wants to spend more time than necessary on a project—but I'd rather have lack of restrictions on application areas and performance. My aim for C++ was and is direct expression of ideas, resulting in code that can be efficient in time and space.

C and C++ have provided stability over decades. That has been immensely important to their industrial users. I have small programs that have been essentially unchanged since the early 80s. There is a price to pay for such stability, but languages that don't provide it are simply unsuitable for large, long-lived projects. Corporate languages and languages that try to follow trends closely tend to fail miserably here, causing a lot of misery along the way.

This leads to thinking about how to manage evolution. How much can be changed? What is the granularity of change? Changing a language every year or so as new releases of a product are released is too ad hoc and leads to a series of de facto subsets, discarded libraries and language features, and/or massive upgrade efforts. Also, a year is simply not sufficient gestation period for significant features, so the approach leads to half-baked solutions and dead ends. On the other hand, the 10-year cycle of ISO standardized languages, such as C and C++, is too long and leads to parts of the community (including parts of the committee) fossilizing.

A successful language develops a community: the community shares techniques, tools, and libraries. Corporate languages have an inherent advantage here: they can buy market share with marketing, conferences, and "free" libraries. This investment can pay off in terms of others adding significantly, making the community larger and more vibrant. Sun's efforts with Java showed how amateurish and underfinanced every previous effort to establish a (more or less) general-purpose language had been. The U.S. Department of Defense's efforts to establish Ada as a dominant language was a sharp contrast, as were the unfinanced efforts by me and my friends to establish C++.

I can't say that I approve of some of the Java tactics, such as selling top-down to nonprogramming executives, but it shows what can be done. Noncorporate successes include the Python and Perl communities. The successes at community building around C++ have been too few and too limited, given the size of the community. The ACCU conferences are great, but why haven't there been a continuous series of huge international C++ conferences since 1986 or so? The Boost libraries are great, but why hasn't there been a central repository for C++ libraries since 1986 or so? There are thousands of open source C++ libraries in use. I don't even know of a comprehensive list of commercial C++ libraries. I won't start answering those questions, but will just point out that any new language must somehow manage the centrifugal forces in a large community, or suffer pretty severe consequences.

A general-purpose language needs the input from and approval of several communities, such as, industrial programmers, educators, academic researchers, industrial researchers, and the open source community. These communities are not disjoint, but individual subcommunities often see themselves as self-sufficient, in possession of knowledge of what is right and in conflict with other communities that for some reason "don't get it." This can be a significant practical problem. For example, parts of the open source community have opposed the use of C++ because "it's a Microsoft language" (it isn't) or "AT&T owns it" (it doesn't), whereas some major industrial players have considered it a problem with C++ that *they* don't own it.

This really crucial problem here is that many subcommunities push a limited and parochial view of "what programming really is" and "what is really needed": "if everybody just did things the right way, there'd be no problem." The real problem is to balance the various needs to create a larger and more varied community. As people grow and face new challenges, the generality and flexibility of a language start to matter more than providing optimal solutions to a limited range of problems.

To get to technical points, I still think that a flexible, extensible, and general static type system is great. My reading of the C++ experience reinforces that view. I am also very keen on genuine local variables of user-defined types: the C++ techniques for handling general resources based on scoped variables have been very effective compared to just about anything. Constructors and destructors, often used together with RAII, can yield very elegant and efficient code.

Teaching

You left industry to become an academic. Why?

Bjarne: Actually, I haven't completely left industry, because I maintain a link to AT&T Labs as an AT&T fellow, and spend much time each year with industry people. I consider my connection with industry essential because that's what keeps my work anchored in reality.

I went to Texas A&M University as a professor five years ago because (after almost 25 years in "The Labs") I felt a need for a change and because I thought I had something to contribute in the area of education. I also entertained some rather idealistic ideas about doing more fundamental research after my years of very practical research and design.

Much computer science research is either too remote from everyday problems (even from conjectured future everyday problems), or so submerged in such everyday problems that it becomes little more than technology transfer. Obviously, I have nothing against technology transfer (we badly need it), but there ought to be strong feedback loops from industrial practice to advanced research. The short planning horizon of many in industry and the demands of the academic publication/tenure race conspire to divert attention and effort from some of the most critical problems.

During these years in academia, what did you learn about teaching programming to beginners?

Bjarne: The most concrete result of my years in academia (in addition to the obligatory academic papers) is a new textbook for teaching programming to people who have never programmed before, *Programming: Principles and Practice Using C++* [Addison-Wesley].

This is my first book for beginners. Before I went to academia, I simply didn't know enough beginners to write such a book. I did, however, feel that too many software developers were very poorly prepared for their tasks in industry and elsewhere. Now I have taught (and helped to teach) programming to more than 1,200 beginners and I feel a bit more certain that my ideas in this area can scale.

A beginner's book must serve several purposes. Most fundamentally, it must provide a good foundation for further learning (if successful, it will be the start of a lifelong effort) and also provide some practical skills. Also, programming—and in general software development—is not a purely theoretical skill, nor is it something you can do well without learning some fundamental concepts. Unfortunately, far too often, teaching fails to maintain a balance between theory/principles and practicalities/techniques. Consequently, we see people who basically despise programming ("mere coding") and think that software can be developed from first principles without any practical skills. Conversely, we see people who are convinced that "good code" is everything and can be achieved with little more than a quick look at an online manual and a lot of cutting and pasting; I have met programmers who considered K&R "too complicated and theoretical." My opinion is that both attitudes are far too extreme and lead to poorly structured, inefficient, and unmaintainable messes even when they do manage to produce minimally functioning code.

What is your opinion on code examples in textbooks? Should they include error/exception checking? Should they be complete programs so that they can actually be compiled and run?

Bjarne: I strongly prefer examples that in as few lines as possible illustrate an idea. Such program fragments are often incomplete, though I insist that mine will compile and run if embedded in suitable scaffolding code. Basically, my code presentation style is derived from K&R. For my new book, all code examples will be available in a compilable form. In the text, I vary between small fragments embedded in explanatory text and longer, more complete, sections of code. In key places, I use both techniques for a single example to allow the reader two looks at critical statements.

Some examples should be complete with error checking and all should reflect designs that can be checked. In addition to the discussion of errors and error handling scattered throughout the book, there are separate chapters on error handling and testing. I strongly prefer examples derived from real-world programs. I really dislike artificial cute examples, such as inheritance trees of animals and obtuse mathematical puzzles. Maybe I should add a label to my book: "no cute cuddly animals were abused in this book's examples."