Software Development for Infrastructure

Bjarne Stroustrup, *Texas A&M University*

Infrastructure software needs more stringent correctness, reliability, efficiency, and maintainability requirements than non-essential applications. This implies greater emphasis on up-front design, static structure enforced by a type system, compact data structures, simplified code structure, and improved tool support. Education for infrastructure and application developers should differ to reflect that emphasis.

Our lives are directly affected by software correctness and efficiency:

- A datacenter, as run by a major corporation such as Amazon, AT&T, Google, or IBM, uses about 15 MW per day (equivalent to 15,000 US homes) and the set-up cost is about US$500 million. In 2010, Google used 2.26 million MW to run its servers.[1]
- A smartphone battery lasts for less than a day if used a lot.
- We're surrounded by systems that, if they fail, can injure people or ruin them economically. Examples include automobile control systems, banking software, telecommunication software, and just about any industrial control software.
- We can't send a repairman to fix a software bug in a space probe, and sending one to reprogram a ship's engine on the high seas is impractical. Trying to fix a software error in a consumer device, such as a camera or a TV set, typically isn't economical.

The implication puts a new emphasis on the old challenge to software developers: deliver code that is both correct and efficient. The benefits achievable with better system design and implementation are huge. If we can double the efficiency of server software, we can make do with one server farm instead of two (or more). If we can double the efficiency of critical software in a smartphone, its battery life almost doubles. If we halve the bug count in safety-critical software, we can naively expect only half as many people to sustain injuries.

This view contrasts with the common opinion that human effort is the factor to optimize in system development and that computing power is essentially free. The view that efficiency and proper functioning are both paramount (and inseparable) has profound implications for system design. However, not all software is the same—it doesn't all "live" under the same conditions and with the same constraints, so we should adjust our software development techniques and tools to take that into account. Similarly, we should adjust our expectations of software developers. One size doesn't fit all when it comes to software, software development, or software developers.

I call software where failure can cause serious injury or serious economic disruption *infrastructure software*. Such software must be dependable and meet far more stringent reliability standards than "regular software." Because ordinary personal computers and smartphones are used as platforms for applications that also serve as infrastructure in my operational use of that word, the fundamental software of such platforms is itself infrastructure, as is the software used to deploy it. For example, if the software that updates the operating system on a cell phone malfunctions, crucial calls might not get through, and someone could be injured or bankrupted.

One of my inspirations for quality infrastructure software is the requirement AT&T placed on switches in its telecommunication system backbone: no more than two hours of downtime in 40 years (www.greatachievements.org/?id=3639). Hardware failure, loss of main power, or a truck colliding with the building that houses the switch isn't an excuse. To reach such goals, we need to become very serious about reliability, which is a vastly different mindset from "we must get something—anything—to market first."

DO MORE WITH LESS

Software reliability is improving. If it were not, we would have been in deep trouble already. Our civilization runs on software. If it were not for computerized systems, most of us would starve. Our dependence on computerized systems is increasing, the complexity of those systems is increasing, the amount of code in those systems is increasing, and the hardware requirements to run those systems are increasing. But our ability to comprehend them is not.

The increases in demands on hardware and software will continue: human expectation grows even faster than hardware performance.

Many of the improvements in system reliability over the years have been achieved through deep layering of software, each layer distrusting other software in the system, checking and rechecking information.[2] Furthermore, layers of software are used to monitor hardware and, if possible, compensate for hardware errors. Often, applications are interpreted or run in virtual machines that intentionally isolate them from hardware. These efforts have resulted in systems that are more reliable, but huge and less well understood.

Another contribution to the improved reliability has come from massive testing. The cost of maintaining and deploying software is increasing. The increases in demands on hardware and software will continue: human expectation grows even faster than hardware performance.

Further consumers of computing power are tools and languages aimed at making programming easier and less error-prone. Unfortunately, many such tools move decisions from design time to runtime, consuming memory and processor cycles. We compensate for lack of design-time knowledge by postponing decisions until runtime and adding runtime tests to catch any errors.

We compensate for our lack of understanding by increasing our requirements for computing power. For example, my word processing software is so "sophisticated" that I often experience delays using it on my dual-core 2.8-GHz computer. But processors are no longer getting faster. The number of transistors on a chip still increases according to Moore's law, but those transistors are used for more processors and more memory. Also, we depend more and more on energy-consuming server farms and on portable "gadgets" for which battery life is an issue. Software efficiency equates to energy conservation.

Reliability and energy efficiency require improvements in many areas. Here, I discuss implications for software. Basically, my conjecture is that we must structure our systems to be more comprehensible. For reliability and efficiency, we must compute less to get the results we need.

Doing more with less is an attractive proposition, but it isn't cost-free: it requires more work up front in the infrastructure software. We must look at high-reliability systems as our model, rather than the techniques used to produce "Internet-time Web applications." Building high-efficiency, high-reliability systems can be slow, costly, and demanding of developer skills. However, this expenditure of time, money, and talent to develop infrastructure hardware isn't just worthwhile, it's necessary. In the longer term, it might even imply savings in maintenance time and costs.

PROGRAMMING TECHNIQUES

I base the discussion here on very simple code examples, but most current infrastructure software doesn't systematically use the techniques I suggest. Code that did so would differ dramatically from what we see today and would be much better for it.

I won't try to show anything radically new, but I highlight what I hope to see deployed over the next 10 years. My examples are in C++, the programming language I know best.[3,4] Of the languages currently used for infrastructure programming, C++ most closely approximates my ideals, but I hope to see even better language and tool support over the next decade. There is clearly much room for improvement.

Compute less

On 23 September 1999, NASA lost its US$654 million *Mars Climate Orbiter* due to a navigation error. "The root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, 'Small Forces,' used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used."[5] The amount of work lost was roughly equivalent to the lifetime's work of 200 good engineers. In reality, the cost is even higher because we're deprived of the mission's scientific results until (and if) it can be repeated. The really galling aspect is that we were all taught how to avoid such errors in high school: "Always make sure the units are correct in your computations."

Why didn't the NASA engineers do that? They're indisputably experts in their field, so there must be good reasons.

No mainstream programming language supports units, but every general-purpose language allows a programmer to encode a value as a `{quantity,unit}` pair. We can encode enough of the ISO standard SI units (meters, kilograms, seconds, and so on) in an integer to deal with all of NASA's needs, but we don't because that would almost double the size of our data. Furthermore, checking the units in every computation would more than double the amount of computation needed.

Space probes tend to be both memory and compute limited, so the engineers—just as essentially everyone else in their situation has done—decided to keep track of the units themselves (in their heads, in the comments, and in the documentation). In this case, they lost. Compilers read neither documentation nor comments, and a magnitude crossed an interface without its unit and suddenly took on a whole new meaning (which was a factor of 4.45 wrong). One conclusion we can draw is that integers and floating-point numbers make for very general but essentially unsafe interfaces—a value can represent anything.

It isn't difficult to design a language that supports SI units as part of its type system. In such a language, all unit checking would be done at compile time and only unit-correct programs would get to execute:

```
Speed sp1 = 100m/9.8s; // fast for a human
Speed sp2 = 100m/9.8s2; // error: m/s2 is acceleration
Speed sp3 = 100/9.8s; // error: speed must be m/s
Acceleration acc = sp1/0.5s; // too fast for a human
```

General-purpose programming languages don't provide direct support for SI units. Some reasons are historical, but the deeper reason is that a language might support a variety of such notions (such as other unit systems, systems for naming dates, markers to help concurrency, and timing constraints). However, we can't build every useful notion into a language, so language designers, programming teachers, and practitioners have preferred the simplicity of doing nothing.

Could a tool or a specialized language supply SI units? In theory, yes, but in practice, specialized languages suffer from high development and maintenance costs and tend not to work well with other specialized tools and languages. Features work best when they're integrated into a general-purpose language and don't need separate tool chains.[6]

Interestingly, a sufficiently expressive language can achieve compile-time unit checking without language extensions. In C++, we can define `Speed` as a simple template that makes the unit (meters per second) part of the type and holds only the quantity as a runtime value. In the recent ISO C++ standard C++11, we can even define literals of those types so that the code fragment above is legal. Doing so isn't rocket science; we simply map the rules of the SI units into the general type system:

```
template<int M, int K, int S>
struct Unit {   // a unit in the MKS system
    enum { m=M, kg=K, s=S };
};

template<typename Unit> // magnitude with unit
struct Value {
    double val;     // the magnitude
    explicit Value(double d)
         : val(d) {} // construct a Value from a double
};

using Speed = Value<Unit<1,0,-1>>;        // m/s
using Acceleration = Value<Unit<1,0,-2>>; // m/s/s

using Second = Unit<0,0,1>;  // s
using Second2 = Unit<0,0,2>; // s*s

constexpr
Value<Second> operator"" s(long double d)
    // a f-p literal suffixed by 's'
{
   return Value<Second> (d);
}

constexpr
Value<Second2> operator"" s2(long double d)
      // a f-p literal  suffixed by 's2'
{
  return Value<Second2 > (d);
}
```

If you aren't familiar with modern C++, much of this code is cryptic. However, it's fundamentally simple and performs all checking and conversions at compile time. Obviously, handling the complete SI unit system takes more code—about three pages in all.

Why bother with the user-defined literals, such as `9.8s`, and `100m`? Many developers dismiss this as redundant and distracting "syntactic sugar." Although a library supporting the SI system has been available in C++ for a decade, very few people have used it. Most engineers and physicists simply refuse to write code using variants like this:

```
// a very explicit notation (quite verbose):
Speed sp1 = Value<1,0,0> (100)/ Value<0,0,1> (9.8);

// use a shorthand notation:
Speed sp1 = Value<M> (100)/ Value<S> (9.8);

// abbreviate further still:
Speed sp1 = Meters(100)/Seconds(9.8);

Speed sp1 = M(100)/S(9.8); // this is getting cryptic
```

Notation matters. SI units are important and should be supported, but so should a variety of other notions. The fundamental point here is that we can improve code quality without adding runtime costs. The use of a static type system improves code quality by reducing the number of errors and moves checking to compile time. In fact, we can move much simple computation to compile time.

Compile-time computation has been done in a variety of languages for decades. Before that (and sometimes still) developers simply precomputed answers and added them to the source code or as data inputs, which is rather ad hoc. In C, compile-time computation implies a rat's nest of complicated and error-prone macros. Such computation is essentially untyped because most information must be encoded as integers. For infrastructure code, I suggest a more systematic and structured approach: type-rich programming at compile time.[7] The SI units example is an illustration.

Compile-time evaluation (and immutability in general) becomes more important as more systems become concurrent: You can't have a data race on a constant.

Access memory less

When I first wrote microcode to squeeze the last bit of efficiency out of a processor, a good rule of thumb was that the system could execute nine instructions while waiting for a memory read to complete. Today, that factor is 200 to 500, depending on the architecture. Memory has become relatively slower. In response, hardware architects have added systems of registers, pipelines, and caches to keep the instructions flowing. This has major implications for software: How do I organize my code and data to minimize memory usage, cache misses, and so on? My first-order answer is

We can improve code quality without adding runtime costs.

- don't store data unnecessarily,
- keep data compact, and
- access memory in a predictable manner.

This again has implications on software design. Consider a simple example: generate *N* random integers and insert them into a sequence so that each is in its proper position in the numerical order. For example, if the random numbers are 5, 1, 4, and 2, the sequence should grow like this:

```
5
1 5
1 4 5
1 2 4 5
```

Once the *N* elements are in order, we remove them one at a time by selecting a random position in the sequence and removing the element there. For example, if we choose positions 1, 2, 0, and 0 (using 0 as the origin), the sequence should shrink like this:

```
1 2 4 5
1 4 5
1 4
4
```

Now, for which *N* is it better to use a linked list than a vector (or an array) to represent the sequence? If we naively apply complexity theory, that answer will be something like, "Is this a trick question? A list, of course!" We can insert an element into and delete from a linked list without moving other elements. In a vector, every element after the position of an inserted or deleted element must be moved. Worse still, if you don't know the maximum number of elements in advance, it's occasionally necessary to copy the entire vector to make room for another element.

Depending on the machine architecture and the programming language, the answer will be that the vector is best for small to medium values of *N*. When I ran the experiment on my 8-Gbyte laptop, I found *N* to be much larger than 500,000. The red line in Figure 1 shows the time taken for the list, and the blue line the time taken by the vector.

This isn't a subtle difference. The *x*-axis is in 100,000 elements, and the *y*-axis in seconds. So for "small lists," a vector is a better representation of a list than a linked structure. This is also true for numbers too small to show on this graph.

Why? First, it takes 4 bytes to store a 4-byte integer in a vector, but it takes 12 bytes to store it in a doubly linked list (assuming 4-byte pointers as links). Saying "list" tripled the memory requirement. Actually, it's worse because many general-purpose list types store each element as a separate object in dynamic (heap, free store) memory, adding another word or two of memory overhead per element. The green line in Figure 1 is a list that I optimized by preallocating space for elements and where each element wasn't a separate object. This demonstrates that even though allocation overheads are significant, they don't explain the vector's fundamental advantage.

Not only are the list nodes large, they're scattered in memory, implying that when we traverse the list to find a position for insertion or deletion, we randomly access memory locations in the area that stored the list, causing cache misses. On the other hand, the hardware really likes the vector's sequential access of words in memory. In an attempt at fairness, I didn't use a binary search to speed up insertion into the vector. Nor did I use random access to find the deletion point in the vector version. This keeps the number of elements traversed the same for all versions. In fact, I used identical generic code for the vector and the lists.

Is this an academic curiosity? No. Infrastructure application developers tell me that compactness and predictable access patterns are essential for efficiency. Power consumption is roughly proportional to the number of memory accesses, so the red (list) and blue (vector) lines are first-order approximations to the drain on a smartphone battery or the number of server farms needed.

We should prefer sequential access of compact structures, not thoughtlessly use linked structures, and avoid general memory allocators that scatter logically related data. We must measure to avoid being blind-sided by unexpected phenomena. Our systems are too complex for us to guess about efficiency and use patterns.

You might say, "But I don't use sequences of 100,000 elements." Neither do I (most of the time), but using 10,000 lists of 100 elements has the same efficiency implications. To developers who write code for huge server farms, 100,000 items are barely noticeable. Consider what's needed to recognize a repeat visitor to a major website and retrieve that person's preferences in time to display a personalized welcome screen. Compactness goes hand in hand with efforts to subdivide larger datasets as well as with the design of algorithms to ensure concurrent execution.[8]

For a small system such as an embedded processor, the differences between compact and linked structures are significant even for small datasets. Even individual object layouts can produce efficiency effects. Consider a simple vector of two-dimensional points:

```
vector<Point> vp = {
    Point{1,2}, Point{3,4}, Point{5,6}, Point{7,8}
};
```

We can represent this in memory as a compact structure with a handle, as in Figure 2, where the blue box represents the overhead required to place memory in dynamic storage. This compact layout is found in a traditional systems programming language, such as C or C++. If necessary, it's possible—at the cost of some flexibility—to eliminate the handle and the dynamic storage overhead.

Alternatively, we can represent the vector as a tree structure, as in Figure 3. This layout is found in a language that doesn't emphasize compact representation, such as Java or Python. The fundamental reason for the difference is that user-defined abstractions (class objects) in such languages are allocated in dynamic memory and accessed through references. The compact representation is 11 words, out of which nine are required data (the eight coordinates plus the number of points). The tree representation is 21 words. In the compact representation, access to a coordinate point requires one indirection; in the tree representation, access requires three indirections.

For languages that are not systems programming languages, getting the compact layout involves avoiding the abstraction mechanisms that make such languages attractive for applications programming.

Practice type-rich programming

So far, I've focused primarily on efficiency, but efficiency is irrelevant if the code contains critical errors. Addressing the issue of correctness requires making progress on two fronts:

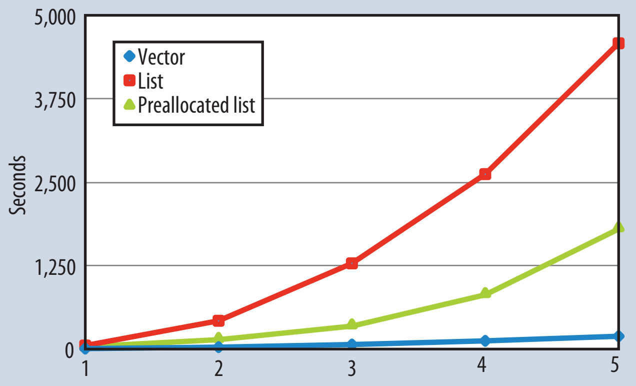

Figure 1. List versus vector timings.

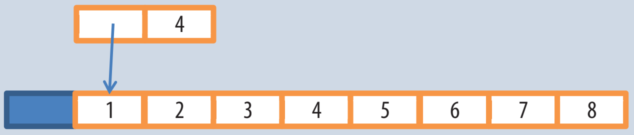

Figure 2. Compact representation.

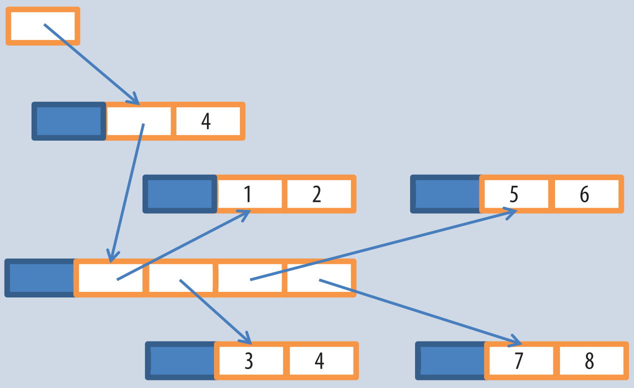

Figure 3. Linked representation.

- eliminate programming errors (make such errors less likely and more easily spotted), and
- make it easier to analyze a program for design errors (in general, make it easier to build tools that manipulate programs).

The distinction between a design error and a programming error isn't completely clear. My practical definition is that programming errors are those that can be caught by a good production compiler; they involve inconsistent use of values and don't require extensive computation to detect. Design errors involve plausible-looking code that just happens to do the wrong thing. Catching design errors requires tools that "understand" the semantics of higher-level operations. Even a compiler can be such a tool if semantic information is encoded in the type system.

Consider

```
void increase_to(double speed);   // speed in m/s
```

This is an error waiting to happen. Maybe we understand the requirements for an argument, but they only appear in comments and other documentation. Is `increase_to(7.2)` correct? Does `7.2` represent a speed? If so, in what units? A better try would be

```
void increase_to(Speed s);
```

Given a reasonable definition of `Speed`, `increase_to(7.2)` is an error and `increase_to(72m/10s)` is likely to be correct. This isn't just an issue of units; I have a philosophical problem with parameters of near-universal types. For example, an integer can represent just about everything. Consider

```
// construct a rectangle:
Rectangle(int x, int y, int h, int w);
```

We want to practice type-rich programming, but we also want to minimize the size of the implementation by using only a few fundamental types.

What does `Rectangle (100,200,50,100)` mean? Are `h` and `w` the coordinates for the bottom right corner or a height and a width? To avoid errors, we should be more specific about `Rectangle`'s requirements on its arguments:

```
Rectangle(Point xy, Area hv);
```

I can now write the clearer and less error-prone

```
Rectangle(Point(100,200), Area(50,100));
```

I can also add a second version:

```
Rectangle(Point top_left, Point bottom_right);
```

and use either of the following to suit my needs and preferences:

```
Rectangle(Point(100,200), Area(50,100));
Rectangle(Point(100,200), Point(150,300));
```

To be relevant for infrastructure development, simple user-defined types (such as `Point` and `Speed`) may not impose overheads compared to built-in types.

Use libraries

Type-rich programming must reflect an overall design philosophy or the software descends into a mess of incompatible types, incompatible styles, and replication. For example, overuse of simple types (such as integers and character stings) to encode arbitrary information hides a program's structure from human readers and analysis tools, but a function specified to accept any `int` will perform its action on integers representing a variety of things. This generality makes functions accepting general types useful in many contexts. Generality makes it easier to design and implement libraries for noncritical uses. It avoids unnecessary replication of effort and code.

We want to practice type-rich programming, but we also want to minimize the size of the implementation by using only a few fundamental abstractions. Object-oriented programming resolves this dilemma by organizing related types into hierarchies, whereas generic programming tackles it by generalizing related algorithms and data structures through (explicit or implicit) parameterization. However, each style of programming handles only a subset of the desirable generalizations: only some relations are hierarchical and only some variation can be conveniently expressed through parameterization. The evolution of languages over the past few years bears this out. For example, Java and C# have added some support for generic programming to their object-oriented cores.

What kinds of libraries are suitable for infrastructure code? A library should encourage type-rich programming to ease human comprehension and tool use, compact data structures, and minimal computation. It should aid in writing efficient and maintainable software, and not encourage bloatware. In particular, it shouldn't "bundle" facilities so that using one library component forces the inclusion of other, potentially unused components. Although the zero-overhead principle—"what you don't use, you don't pay for"—is remarkably difficult to follow when designing tools and systems, it's extremely worthwhile as a goal. Composing solutions to problems out of separately developed library and application components should be easy and natural.

Many aspects of libraries are determined by the type systems of their implementation languages. A language choice determines the overheads of the basic operations and is a major factor in the style of libraries. Obviously, library design for infrastructure is a more suitable topic for a series of books than a section of an article, but short examples can illustrate the role of types.

I see a type system primarily as a way of imposing a definite structure—a technique for specifying interfaces so that a value is always manipulated according to its definition. Without a type system, all we have are bits, with their meaning assigned by any piece of code that accesses them. With a type system, every value has a type. Moreover, every operation has a type and can accept only operands of its required argument types. It isn't possible to realize this ideal for every line of code in every system because of the need to access hardware and communicate between separately developed and maintained systems. Furthermore, backward compatibility requirements for languages and protocols pose limitations. However, type safety is an unavoidable ideal. Encoding a program's static structure in the type system (ensuring that every object has a type and can hold only the values of its type) can be a major tool for eliminating errors.

Programmers can and do vigorously disagree about the meaning of "type" and its purpose. I tend to emphasize a few significant benefits:

- more specific interfaces (relying on named types), implying early error detection;
- opportunities for terse and general notation (such as + for addition for any arithmetic type or `draw()` for all shapes);
- opportunities for tool building that rely on high-level structure (test generators, profilers, race condition finders, and timing estimators); and
- improved optimization (for example, references to objects of unrelated types can't be aliases).

We can classify widely used languages by their support for types:

- *Languages that provide only a fixed set of types and no user-defined types (for example, C and Pascal).* Records (structs) provide representations for composite values and functions provide operations. Popular built-in types (such as integers, floating-point numbers, and strings) are overused to specify interfaces with no explicit high-level semantics. A trivial type system can catch only trivial errors.
- *Languages that provide user-defined types (classes) with compile-time checked interfaces (such as Simula, C++, and Java).* They also tend to support runtime polymorphism (class hierarchies) for added flexibility and extensibility. Very general interfaces (for example, `Object`) are often overused to specify interfaces with no explicit high-level semantics. Semantically meaningful operations, such as initialization and copy can be associated with user-defined types.
- *Languages that provide user-defined types (classes) with runtime type checking (such as Smalltalk, JavaScript, and Python).* An `Object` can hold values of any type. This implies overly general interfaces.

The demands of correctness and efficiency will push infrastructure developers toward a disciplined use of the second alternative: rich, extensible type systems with named, statically checked, and semantically meaningful interfaces.

Alternative one, writing code without serious use of user-defined types, leads to hard-to-comprehend, verbose results with few opportunities for higher-level analysis. However, this kind of code (usually written in C or low-level C++) is popular because it's easy to teach the language basics (the complexities disappear into the application code) and provide low-level analysis.

There's a widespread yet mistaken belief that only low-level, messy code can be efficient. I once gave a presentation of a C++ linear-algebra library that achieved astounding efficiency because it used a type system that allowed it to eliminate redundant temporaries and apply optimized operations by "knowing" (from the static type system) the fundamental properties of some matrices.[9] Afterward, I was repeatedly asked, "But how much faster would it run if it was rewritten in C?" Many developers equate "low level" with "fast" out of naiveté or from experience with complicated bloatware.

Relying heavily on runtime resolution or interpretation does not provide the maintainability and efficiency needed for infrastructure.

Alternative three, relying heavily on runtime resolution or interpretation, doesn't provide the maintainability and efficiency needed for infrastructure. Too many decisions are hidden in conditional statements and calls to overly general interfaces. Obviously, I'm not saying that JavaScript (or whatever) is never useful, but I do suggest that the JavaScript engine should be written in a language more suitable for systems programming (as it invariably is).

One of the advantages of dynamic typing is that it (typically) provides "duck typing" ("if it walks like a duck and quacks like a duck, it's a duck," or, in more technical terms, values have types, but objects do not—an object's behavior is determined by the value it holds). This can be used to provide more general and flexible libraries than interface-based class hierarchies. However, duck typing is suspect in infrastructure code; it relies on weakly specified, very general interfaces. This can result in unexpected semantics and need for runtime checking. It simplifies debugging but complicates systematic testing. Runtime typing carries heavy costs—often a factor of 10 to 50 (http://shootout.alioth.debian.org): objects are larger because they must carry type information, resolving types at runtime implies extra computation compared to less dynamic alternatives, and optimization is inhibited because interfaces must be able to handle objects of every type, if only to give an error message.

However, a statically typed language can provide much of the desired flexibility. In particular, duck typing, as provided by templates, is the backbone of generic programming in C++. It's also the practical basis of the ISO C++ standard library and most high-efficiency C++ (including generic code optimized for particular hardware through parameters). For example,

```
template<typename Container, typename Predicate>
typename Container::iterator
find_if(Container& c, Predicate pred)
     // return an iterator to the first element in c
     // for which pred(element) is true
{
      auto p = c.begin();
      while(p!=c.end() && !pred(*p)) ++p;
      return p;
}

void user (vector<string>& v,
    list<Record*>& lst,
    Date threshold)
{
      auto p = find_if(v,
               [](const string& s)
                       { return s<"Fulcrum"; }
         );
      // …
      auto q = find_if(lst,
               [](const Record* p)
                       { return threshold<p->date; }
)
    // …
}
```

Errors that manifest as exceptions in a dynamically checked language become compile-time errors.

Here, the generic function `find_if()` is called with dramatically different arguments: a `vector` of `strings`, and a `list` of pointers to `Records`. The search criteria are presented as lambda expressions. Every modern C++ compiler can generate code for this that equals or outperforms elaborate hand-coded alternative implementations.[10] Note the way that code from different sources is used to compose the solution:

- the `list`, `vector`, and `string` from the ISO C++ standard library;
- the `find_if()` and `Record` from my application; and
- the lambda expressions written specifically in `user()`.

There's no need for "glue code," use of inheritance, wrapping of built-in types (such as `Record*`) into self-describing objects, or dynamic resolution. This style of generic code is often called STL-style.[11]

However, the interfaces are still underspecified—in particular, the definition of `find_if()` didn't say that its first argument had to be a `Container`. Errors that manifest as exceptions in a dynamically checked language become compile-time errors. Unfortunately, the reporting of those errors can be very obscure. Consequently, much effort in modern C++ is aimed at better specification of template arguments.[12,13] We also need to add semantic constraints[14]—for example, a generic function that uses a copy operation needs to know whether that copy is deep or shallow. Similarly, a generic function that uses a + needs to know whether that + is an arithmetic function or something else, such as a concatenation.

An explicit representation of system structure can simultaneously increase flexibility, improve error diagnostics, and improve efficiency. Correctness and efficiency are closely related properties of a well-specified system. It seems that extensive use of a rich type system leads to shorter code independently of whether the type system is static or dynamic. Obviously, we can also use types to gain the usual aspects of an object-oriented system—encapsulation, runtime polymorphism, and inheritance—but we can't go overboard and try to rely completely on static structure. That isn't necessary for noncritical parts of a system, and a complete absence of runtime checks would leave the system open to catastrophic errors caused by malfunctioning hardware. Dealing gracefully with hardware failure is a crucial characteristic of infrastructure systems.

Prefer highly structured code

It isn't enough to be disciplined in our specification of data structures and interfaces: we must also simplify our code logic. Complicated control structures are as dangerous to efficiency and correctness as are complicated data structures.

When choosing among alternatives in code, we often have three options:

- *Overloading.* Select among alternative functions based on the static type of arguments, for example, `f(x)`.
- *Virtual function call.* Select based on the dynamic type of a class object in a class hierarchy, for example, `x.f()`.
- *Selection statement.* Select on a value with an if-statement or a switch-statement, for example, `if (x.c) f1(x) else f2(x)`.

Using an if-statement is the least structured and most flexible option. However, we should choose the more structured and easier-to-optimize alternatives whenever possible. The hierarchy option can be most useful, but was seriously overused in the past couple of decades. Not every class naturally belongs in a hierarchy, not every class benefits from the coupling that a hierarchy introduces, and not every class is improved by the possibility of adding to it through derivation (subclassing). Of the alternatives, a call of an ordinary function is the easiest to understand—it's also often easy to inline. Consequently, overloading—statically selecting a function based on argument types—can lead to major efficiency advantages compared to indirect calls (as used to select among alternatives in a class hierarchy).

To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs.

Consider a real example (taken from Sean Parent): in many user interfaces, an item of a sequence can be moved to a different position in the sequence by dragging its screen representation over a screen representation of the sequence. The original solution (after cleanup and simplification) involved 25 lines of code with one loop, three tests, and 14 function calls. Its author considered it messy and an attempt to simplify the code was made. The result was

```
void drag_item_to(Vector& v,
                  Vector::iterator source,
                  Coordinate p)
{
   auto dest = find_if(v.begin(), v.end(), contains(p));
    if (source < dest)
        // from before insertion point:
        rotate(source, source+1, dest);
    else
        // from after insertion point:
        rotate(dest, source, source+1);
}
```

That is, find the insertion point using the standard library algorithm `find_if` and then move the element to the insertion point using the standard library algorithm `rotate`. Obvious once you see it!

The original solution was complicated enough to raise worries about correctness, plus it was completely special-purpose. What might happen to such hard-to-understand code during maintenance? The improved code is shorter, simpler, more general, and runs faster. It uses only well-documented standard library facilities known to experienced C++ developers.

But why move only one element? Some user interfaces allow selecting and moving a collection of elements. And why use application-specific notions such as `Vector` and `Coordinate`? Dealing with this more general problem turns out to be simpler still:

```
template <typename Iter, typename Predicate>
pair<Iter, Iter>
gather(Iter first, Iter last, Iter p, Predicate pred)
// move e for which pred(e) to the insertion point p
{
     return make_pair(
        // from before insertion point:
        stable_partition(first, p, !bind(pred, _1)),
        // from after insertion point:
        stable_partition(p, last, bind(pred, _1))
      );
}
```

The `Predicate` determines which elements are moved. Admittedly, `stable_partition` tends to be used only by specialists, but its meaning isn't obscure: `stable_partition(first,last,pred)` places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that don't satisfy it. Note the absence of explicit tests and loops.

Expressing code in terms of algorithms rather than hand-crafted, special-purpose code can lead to more readable code that's more likely to be correct, often more general, and often more efficient. Such code is also far more likely to be used again elsewhere. However, making algorithm-based code mainstream requires a culture change among systems programmers. Analyzing problems with the aim of coming up with general, elegant, and simple solutions isn't emphasized in the education of or selection of systems programmers. Too many programmers take pride in complicated solutions (invariably assumed to be efficient), and too many managers are more impressed with such code than with simpler alternatives.

Complicated control flows also complicate resource management. A resource is anything that has to be acquired and (explicitly or implicitly) released after use. Memory, locks, sockets, and thread handles are examples.[15] For efficiency and local reasoning, it's preferable to hold a resource for as short a time as possible and for that to be obvious.

Consider a slightly artificial simple example. The function `algo` updates all records from a `vector` that matches a predicate and returns a list of indices of the updated records:

```
vector<int>  algo(vector<Record>& vr, Predicate pred)
   // update vr[i] if pred(vr[i])
{
      vector<int> indices;   // updated records
      for (int i = 0; i<v.size(); ++i)
        if (pred(vr[i]))  {
           unique_lock lck(vr[i].mtx); // acquire mutex
          // update vr[i]
          indices.push_back(i); // record the update
        }
      return indices;
}
```

Here, I assume concurrency, so that some form of mutual exclusion is necessary. The `vector` and `unique_lock` come from the C++ 11 standard library; `Record` and `Predicate` are assumed to be defined by the application. Two resources are required here: the memory for elements of the `vector` of indices, and the `mutex` members (`mtx`). The lifetime of the `unique_lock` determines how long its `mutex` is held. Both `vector` and `unique_lock` hold onto their resource while it's in scope and release it when that scope is exited.

Note that there is no user code for handling error returns from `algo`: `vector` and `unique_lock` release their resources even if an exit from their scope is done by a return or by throwing an exception. Explicit resource management in the presence of errors can be a major source of complexity—for example, nested conditions or nested exception handlers—and obscure errors. Here, the resource management is made part of the semantics of the types used (`vector` and `unique_lock`) and is therefore implicit. Importantly, the resource management is still local and predictable: it follows the lexical scope rules that every programmer understands.

Naturally, not all resource management can be scoped and not all code is best expressed as highly stylized algorithms. However, it's essential to keep simple things simple. Doing so leaves us with more time for the more complex cases. Generalizing all cases to the most complex level is inherently wasteful.

WHY WORRY ABOUT CODE?

Do we need traditional programmers and traditional programs? Can't we just express what we want in some high-level manner, such as pictures, diagrams, high-level specification languages, formalized English text, or mathematical equations, and use code-generation tools to provide compact data structures and fast code? That question reminds me of an observation by Alan Perlis: "When someone says, 'I want a programming language in which I need only say what I wish done,' give him a lollipop" (www.cs.yale.edu/quotes.html).

True, we've made progress—Modelica, for example (https://modelica.org)—but generative techniques work best for well-understood and limited application domains, especially for domains with a formal model, such as relational databases, systems of differential equations, and state machines. Such techniques have worked less well in the infrastructure area. Many tasks aren't mathematically well-defined, resource constraints can be draconian, and we must deal with hardware errors.

Many tasks aren't mathematically well-defined, resource constraints can be draconian, and we must deal with hardware errors.

I see no alternative to programmers and designers directly dealing with code for most of these tasks. Where something like model-driven development looks promising, the generated code should be well-structured and human readable—going directly to low-level code could easily lead to too much trust in nonstandard tools. Infrastructure code often "lives" for decades, which is longer than most tools are stable and supported.

I'm also concerned about the number of options for code generation in tools. How can we understand the meaning of a program when it depends on 500 option settings? How can we be sure we can replicate a result with next year's version of the tool? Even compilers for formally standardized language aren't immune to this problem.

Could we leave our source code conventionally messy? Alternatively, could we write code at a consistently high level isolated from hardware issues? That is, could we rely on "smart compilers" to generate compact data structures, minimize runtime evaluation, ensure inlining of operations passed as arguments, and catch type errors from source code in languages that don't explicitly deal with these concepts?

These have been interesting research topics for decades, and I fear that they will remain so for even more decades. Although I'm a big fan of better compilers and static code analysis, I can't recommend putting all of our eggs into those baskets. We need good programmers dealing with programming languages aimed at infrastructure problems.

I suspect that we can make progress on many fronts, but for the next 10 years or so, relying on well-structured, type-rich source code is our best bet by far.

THE FUTURE

Niels Bohr said, "It is hard to make predictions, especially about the future." But, of course, that's what I've done here. If easy-to-use processing power continues to grow exponentially, my view of the near future is probably wrong. If it turns out that most reliability and efficiency problems are best solved by a combination of lots of runtime decision-making, runtime checking, and a heavy reliance on metadata, then I have unintentionally written a history paper. But I don't think I have: correctness, efficiency, and comprehensibility are closely related. Getting them right requires essentially the same tools.

Low-level code, multilevel bloated code, and weakly structured code mar the status quo. Because there's a lot of waste, making progress is relatively easy: much of the research and experimentation for many improvements have already been done. Unfortunately, progress is only relatively easy; the amount of existing code and the number of programmers who are used to it seriously complicate any change.

Hardware improvements make the problems and costs resulting from isolating software from hardware far worse than they used to be. For a typical desktop machine,

- 3/4ths of the MIPS are in the GPU;
- from what's left, 7/8ths are in the vector units; and
- 7/8ths of that are in the "other" cores.

So a single-threaded, nonvectorized, non-GPU-utilizing application has access to roughly 0.4 percent of the compute power available on the device (taken from Russell Williams). Trends in hardware architecture will increase this effect over the next few years, and heavy use of software layers only adds to the problem.

I can't seriously address concurrency and physical distribution issues here, but we must either find ways of structuring infrastructure software to take advantage of heterogeneous, concurrent, vectorized hardware or face massive waste. Developers are already addressing this problem with careful programming using architecture-specific concurrency primitives for the code closest to the hardware and using threads libraries for system-level concurrent tasks. Getting such code correct, efficient, maintainable, and portable to next year's hardware adds to the requirements for structure, compactness, and code formalization (algorithms).

Locality and compactness are key to comprehensibility and efficiency for nonuniform processors and memory. We need to be able to reason about code without knowledge of the complete system. Shared resources are poison to such reasoning, making local resource management essential.

We also need more and better tools for specifying requirements, analyzing code, verifying system properties, supporting testing, measuring efficiency, and so on. However, we shouldn't expect miracles: designing, implementing, and using such tools tends to be difficult.[16-20] My conjecture is that much of the complexity of such tools comes from the messiness of the code they're supposed to deal with. Whatever else we do, we must also clean up our code.

What should be done? There's a saying that "real software engineers write code." I wish that were true. Production of quality code should be elevated to a central role in software development. A software developer should be able to proudly display a portfolio of work (including code), much the way an artist or architect does. We should read and evaluate code as part of every project. We should distinguish between infrastructure code and application code. Often, the two areas need different languages, tools, and techniques. Sometimes, that's the case even when we use the same language for both infrastructure and applications. The role of static typing should be increased.

All of this has implications for education: you can't expect a person trained to deliver applications quickly in JavaScript to design and implement an infrastructure component (say, a JavaScript engine or a JVM) in C++ using the same skills. We need to specialize at least part of our computer science, software engineering, and IT curricula.[21] I strongly prefer general-purpose programming languages, but no single language is ideal for everything. We should master and use multiple languages, often as part of a single system.

Infrastructure developers should be highly skilled professionals, not assembly-line workers, and not just scholars. The mathematical basis of their work must be strengthened. We need developers with good analytical skills, some ability in algebraic reasoning (calculus alone isn't sufficient), and enough familiarity with statistics to understand systems through measurement. Obviously, algorithms, data structures, machine architecture, and operating systems must remain core subjects to provide a basis for emphasizing efficiency and reliability.

In research, we need a greater appreciation of incremental (engineering) improvements with a relevance to real-world systems. "That's just engineering, and we're computer scientists" is an attitude we can't afford. I suspect the era of transformative breakthroughs is over. We need to achieve a factor-of-two-or-three improvement in several areas, rather than trying to solve our problems with a single elusive two-orders-of-magnitude improvement from a silver bullet. C

References

1. R. Waters, "Server Farm Cost," *Financial Times*, 9 Sept. 2011; www.ft.com/cms/s/2/6e358ae0-da7a-11e0-bc99-00144feabdc0.html#axzz1YhypogYG.
2. N. Mitchell et al., "The Diary of a Datum: An Approach to Analyzing Runtime Complexity," *Proc. Library-Centric Software Design*, ACM, 2006; http://domino.watson.ibm.com/comm/research_projects.nsf/pages/sorbet.pubs.html/$FILE/diaryslides.pdf.
3. B. Stroustrup, *The C++ Programming Language* (special ed.), Addison-Wesley, 2000.
4. "Programming Languages—C++," Int'l Standard ISO/IEC 14882-2011; www-d0.fnal.gov/~dladams/cxx_standard.pdf.
5. "Mars Climate Orbiter Mishap Investigation Board Phase I Report," NASA, 10 Nov. 1999; ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.
6. B. Stroustrup, "A Rationale for Semantically Enhanced Library Languages," *Proc. Library-Centric Software Design* (LCSD 05), Rensselaer Polytechnic Inst., 2005; www.cs.rpi.edu/research/pdf/06-12.pdf.
7. G. Dos Reis and B. Stroustrup, "General Constant Expressions for System Programming Languages," *Proc. Symp. Applied Computing* (SAC 10), ACM, 2010; http://www2.research.att.com/~bs/sac10-constexpr.pdf.
8. A.A. Stepanov et al., "SIMD-Based Decoding of Posting Lists," *Proc. ACM Conf. Information and Knowledge Management* (CIKM 11), ACM, 2011; www.stepanovpapers.com/CIKM_2011.pdf.
9. J. Siek, A. Lumsdaine, and L.-Q. Lee, "Generic Programming for High-Performance Numerical Linear Algebra," *Proc. SIAM Workshop Interoperable Object-Oriented Scientific Computing*, SIAM, 1998, pp. 1-10; www.osl.iu.edu/publications/prints/1998/siek98:_siamoo.pdf.
10. B. Stroustrup, "Learning Standard C++ as a New Language," *C/C++ Users J.*, May 1999; http://www2.research.att.com/~bs/new_learning.pdf.
11. A. Stepanov and M. Lee, *The Standard Template Library*, tech. report 95-11(R.1), HP Labs, 1995.
12. G. Dos Reis and B. Stroustrup, "Specifying C++ Concepts," *Proc. Symp. Principles of Programming Languages* (POPL 06), ACM, 2006; http://www2.research.att.com/~bs/popl06.pdf.
13. D. Gregor et al., "Concepts: Linguistic Support for Generic Programming in C++," *Proc. Object-Oriented Programming,*

Systems, Languages & Applications (OOPSLA 06), ACM, 2006; http://www2.research.att.com/~bs/oopsla06.pdf.

14. A. Sutton and B. Stroustrup, "Design of Concept Libraries for C++," *Proc. Software Language Eng.* (SLE 11), 2011; http://www2.research.att.com/~bs/sle2011-concepts.pdf.
15. B. Stroustrup, "Exception Safety: Concepts and Techniques," LNCS 2022, Springer, 2001; http://www2.research.att.com/~bs/except.pdf.
16. A. Bessey et al., "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Comm. ACM*, Feb. 2010, pp. 66-75; http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext.
17. G. Dos Reis and B. Stroustrup, "A Principled, Complete, and Efficient Representation of C++," *J. Mathematics in Computer Science*, 2011; http://www2.research.att.com/~bs/macis09.pdf.
18. D. Parnas, "Really Rethinking Formal Methods," *Computer*, Jan. 2010, pp. 28-34; www.computer.org/portal/web/csdl/doi/10.1109/MC.2010.22.
19. T. Ramananandro, G. Dos Reis, and X. Leroy, "Formal Verification of Object Layout for C++ Multiple Inheritance," *Proc. Symp. Principles of Programming Languages* (POPL 11), ACM, 2011; http://gallium.inria.fr/~xleroy/publi/cpp-object-layout.pdf.
20. G.J. Holzmann, *The Spin Model Checker*, Addison-Wesley, 2003.
21. B. Stroustrup, "What Should We Teach Software Developers? Why?" *Comm. ACM*, Jan. 2010, pp. 40-42; http://cacm.acm.org/magazines/2010/1/55760-what-should-we-teach-new-software-developers-why/fulltext.

***Bjarne Stroustrup**, the College of Engineering Chair Professor in Computer Science at Texas A&M University, is the designer and original implementer of C++ and the author of several books, including* Programming—Principles and Practice Using C++ *and* The C++ Programming Language. *His research interests include distributed systems, programming techniques, software development tools, and programming languages. Stroustrup is actively involved in the ISO standardization of C++. He is a member of the US National Academy of Engineering and a fellow of IEEE and ACM. He retains a link to AT&T as an AT&T Fellow. Stroustrup received a PhD in computer science from Cambridge University, England. Contact him at bs@cs.tamu.edu; www.research.att.com/~bs.*

Selected CS articles and columns are available for free at http://ComputingNow.computer.org.

THERE NOW IS A **QUICK AND EASY** WAY TO FIND OUT ABOUT OUR **COLLECTION OF TRANSACTIONS**

Our new **"What's New in Transactions"** webpage provides an overview of our 14 peer-reviewed scholarly journals.

- Read free articles from each title,
- View and listen to new multimedia,
- Learn about recent special issues and special sections, and
- Discover open calls for papers.

It is ever changing like the research areas our journals cover, so make sure to come back frequently. Visit **http://www.computer.org/whats-new** today!

IEEE   IEEE computer society