GETTING FROM THE PAST TO THE FUTURE

Bjarne Stroustrup

THE MAIN PROBLEM for the C++ community today is to use Standard C++ in the way it was intended rather than as a glorified C or a poor man's Smalltalk.

Standard C++ offers a balanced set of facilities for efficient, flexible, and type-safe programming; modern implementations offer reasonable—and steadily improving—support for them. However, some practical concern and much propaganda discourage people from using these facilities. For example, dire warnings against the evils of multiple inheritance, templates, exceptions, and the STL are not uncommon. What's worse, "coding standards" and "style rules" selectively demonize language features introduced after 1987.

To make major progress and to leave many of the problems of the past behind, we must take a different approach. We must emphasize higher-level programming styles and the language features that support them. We must also consider which older styles of programming and which language features have become serious drags on our ability to build efficient and maintainable systems.

I don't believe in simple "thou shalt not" style coding rules, but if I did, I'd suggest we start with features that make legacy code hard to deal with. For example:

- Never expose a `T**` to an application programmer.
- Never require an application programmer to write a cast.
- Never expose a `void*` to an application programmer.

I would also have some strong words to say about macros—including macros used to hide such crud. I'm sure that you have seen libraries and recommended coding practices that force you to break these rules almost everywhere. Those coding practices are major sources of bad software. For our purposes, "bad" means unreadable, unmaintainable, unportable, and inefficient.

The alternative is to focus on programming and design technique rather than language features, and use language features that allow us to express designs directly in code; for example: containers for holding objects and values; templates for generic programming and type-safe interfaces; concrete types for simple objects; class hierarchies, preferably based on abstract classes, for generalizations where exact types cannot be known until runtime; and exceptions for error handling.

We cannot afford to be starry-eyed idealists. There are millions of lines of ugly but useful code out there, no shortage of C++ implementations that are not up to the level of the standard, and no shortage of interfaces that expose low-level details. However, I don't mind legacy *code*, I mind legacy *interfaces*. The problem is not the old code but the insistence on making new code according to old rules, rather than designing and programming using better techniques and providing clean interfaces to unclean code.

The future belongs to people writing code using these techniques. The future belongs to people writing code that is type-safe, uses templates and exceptions, and relies on

extensive libraries of concrete types and class hierarchies. I'd like those programmers to use C++ because Standard C++ currently provides a better balance of support for these techniques than any other major programming language. However, the best code will be written by people who use these techniques and language features—in whatever language—rather than people who insist on writing Fortran, K&R C, etc., in whichever language they (mis)use.

Pay due respect to the lessons of the past and the present realities, but face the future. The future will be great; see you there!

Bjarne Stroustrup is the designer and original implementor of C++ and the author of The C++ Programming Language *(1st edition 1985, 2nd edition 1991, 3rd edition 1997) and* The Design and Evolution of C++. *He is the head of AT&T Labs' Large-Scale Programming Research deparment and an AT&T Fellow. He can be contacted at bs@research.att.com.*

KNOW WHAT YOU KNOW

By Andrew Koenig

> HOW MANY TIMES have you seen someone debug a program this way:
>
> 1. Find a part of the program that looks like it might contain the bug that you're seeking.

2. Change something at random. Try the program.

3. If it works, you're done! Otherwise, go back to step 1.

How many times have you seen someone figure out how to use a complicated library interface this way:

- Find a part of the interface that looks like it might do what you want.
- Make a guess about how to call it. Try it out.
- If it works, you're done! Otherwise go back to step 1.

Of course, even if all of our friends program this way, we know better, right? (He asked, tongue firmly planted in cheek.)

We all know better than to program and debug this way. So why do we continue to do it? I think the answer comes from three factors that have dominated computing over the past 20 years: speed, complexity, and interaction.

In 1980, a typical departmental minicomputer offered about 1 MIPS, and a personal computer about 0.3 MIPS. Today's machines are about a thousand times that. System software hasn't grown in size that quickly, but today's systems are still huge by earlier standards, and the amount of information needed to understand them is daunting. At the same time, systems have become more interactive, so experimentation has become easier.

Together, these trends foster a dangerous mind-set in programmers:

> *Dangerous approach: Don't try to understand how the system works-just try it and see what happens.*

The result of this mind-set is software that seems to work, but that collapses under load, or fails to repel intruders, or gives wrong answers that are just good enough to escape notice for a while.

As programmers, we can resist this mind-set by adopting a simple philosophy:

> *Know, don't guess.*

Most of us know when we're sure we know something. When we are unsure, we should look it up or ask someone who knows, rather than rely on experiments. Experiments can confirm our understanding, and can even prove failure, but they cannot prove success.

I am not suggesting that programmers should always know everything about every tool they use—just that we must know what we know and what we don't know, and verify information that we need whenever we are less than certain. We need a core of confident knowledge about our most important tools and a source of accurate information about what we don't know for certain.

These needs tell us how to design and document our systems: They must have a clear conceptual foundation, which we must use as the basis for describing how those systems work. In particular, the common practice of documenting a system by listing recipes for solving typical problems is a good way to get in trouble, because the lack of a clear model encourages users to guess rather than to understand.

These needs also tell us how to teach programming. Instead of flooding students with tiny details, we should concentrate on confident knowledge about the fundamentals, along with an attitude of "Here is where to look up this other stuff if you need it."

This attitude may seem old-fashioned. However, as the number and complexity of available tools continues to grow, there is less and less of a choice.

The computing world has grown enormously, and will continue to grow for the foreseeable future. To cope with that growth, we must realize that we cannot know all there is to know. This incomplete knowledge is not an insurmountable problem, as long as we know enough to be able to distinguish what we know from what we don't.

Andrew Koenig is a Principal Technical Staff Member at AT&T Research and Project Editor of the ISO/ANSI C++ standards committee. He can be contacted at ark@research.att.com.

DIVERSIFY

By John Vlissides

A COMMON AND generally unwelcome by-product of mastering a skill is the

narrowing effect. I'm not talking about getting thinner; I'm talking about how the single-minded pursuit of a skill affects other areas of your professional life. Mastery has an opportunity cost: You focus on something so intensely that other skills go to seed, or you never acquire them in the first place.

Expertise can become an end in itself. It has a powerful allure, especially when there isn't enough of it to go around. A skill vacuum always accompanies hot topics—the Javas, UMLs, and components of this era or that. What about C++, you ask? That's the thing about vacuums: They can be self-perpetuating. Even though C++ is not the hot property it used to be, the new ANSI Standard guarantees a dearth of C++ expertise well into the next century. Heck, more than 30 years have passed since Simula gave us objects, and most people still aren't comfortable with them.

The relevance to the C++ programmer is twofold. First, don't spend your life learning every last detail of C++—or the UML, or the methodology *du jour*. Don't limit yourself to pure design, pure implementation, or pure analysis. Leave extreme specialization to the medical profession; ours is too new, too dynamic to focus on any area exclusively. It's crucial to know your tools and to use them well, but don't obsess over them.

There are people who have fun studying the C++ spec for literally seconds on end, I'm sure, but trying to absorb it all is just plain futile. There's no way you're going to appreciate the ins and outs of every language feature without applying them in practice—and by the way, I know not one soul who has exercised all of C++ in practice. There can be value in exploring some nook or cranny even if you never use it, I'll admit. You just have to balance this value against that of learning something relevant outside C++.

When you confront a real problem that resists customary solutions, it makes sense to go on a feature hunt. Even then, chances are that someone, somewhere, has faced that problem before, possibly in a different guise. Seek out the existing solution. Is it written up as a pattern, perchance? Consult the Pattern Almanac.[1] Browse the Portland Pattern Repository[2] or other online pattern resources. Post a query to a newsgroup or mailing list.[3] When all else fails, go to the language spec and craft a solution from first principles; until then, resist the urge to reinvent the wheel.

Second, spend the time you just saved on other areas of learning. Broaden your horizons. Delve into Java, XML, even (heaven forbid!) Visual Basic. Branch out. Learn about the special needs of software for low-power devices such as PDAs and cell phones. Get crazy and study organizational management. Curl up with that long-shelved novel. Read to your kids, for pete's sake. A gear-change can do wonders for your creativity, not to mention the insights that can come from a little cross-pollination.

There *is* life beyond C++. Don't let it pass you by.

John Vlissides is a member of the research staff at IBM's T.J. Watson Research Center in Hawthorne, NY. He can be contacted at vlis@watson.ibm.com.

FORGET C++!

By Jesse Liberty

LAST YEAR I wrote that the best way to learn C++ is not to focus on the code

but rather to focus on the design. This year I go one step further: The best way to learn C++ is to stop being in love with it. A friend told me once, "No one ever wanted a 3/8 drill bit. They wanted a 3/8 inch hole." There are no greater words of insight for a programmer. While C++ is still the premier language for commercial software development, let's not confuse the means with the ends: The goal is great products. The trick is to focus on the semantics of what you're trying to build rather than on the specific syntactic sugar that C++ happens to require. To do this you must look beyond the details of the language to the model that you're trying to implement.

When I wanted to learn photography, I asked an award-winning photographer which camera to buy. I was hoping he'd suggest a top-end Nikon. His suggestion was a fixed-focus, bottom-of-the-line Instamatic. His advice: Shoot with nothing but this for six months. Stop thinking about F-stops and shutter speed and pay attention to shadow and composition. When you're done, you'll know a lot more about photography, and then all the gizmos won't get in your way of seeing the picture. As programmers, we must remember to pay attention to the design and not get too hung up on the syntax of the specific language we happen to be programming in right now. C++ is great, but it isn't an end in itself. This is why analysis (understanding the requirements) and design (modeling a solution) are so important. Implementation is, in many ways, the easy part. The hard part is helping your client understand what he really wants and needs and then designing a robust, reliable, scalable, and maintainable solution. What makes you a valuable programmer is your ability to think beyond the code.

Jesse Liberty is the author of numerous books on C++ and object-oriented software development, including C++ from Scratch *and* Beginning Object-Oriented Analysis and Design. *Please visit his Website at www.LibertyAssociates.com.*

INCREMENTAL ARCHITECTURE CAPTURE

By James Grenning

AS SOFTWARE ENGINEERS, many of us are often faced with successful software products that are built on the software equivalent of a house of cards. By "successful," I mean customers want the product, pay good money for it, and pay more good money to have it change and evolve. Because of market pressures, inexperience, or just the relentless influence of entropy, the internal system architectures have deteriorated. What once was an elegant system is now a mess internally. The product may still be viable, if customers want to buy it and pay for upgrades, but it may be expensive.

Faced with this situation, we have a few choices: Scrap the system and completely redevelop it; continue to hack in new changes and suffer the consequences; or incrementally improve the system architecture (there are probably other choices as well). Scrapping the system sounds good to developers; they love clean sheet designs. The marketing and management people cringe at this thought, "You want to spend *n* man years just to get back to today's functionality?" If the product is thought to have a very long future, and the market can support the redevelopment, this may be a viable alternative. Usually it is not.

Hacking new features into an old system is the least favorite task of the development team and has its own problems with side-effect defects, unpredictable schedules, and development effort. Again, depending upon your situation, this may be the route to take. Is

there a limited lifetime? Are the changes few and far between? Can we afford the additional support that the unreliability will cause?

These are not the alternatives I really want to talk about. In legacy software, a defined architecture is often foreign. The software architecture may have been lost years earlier as the parade of requirement changes barrage the system. If your product is still expecting a long life, but a clean sheet design is not practical, an incremental architecture improvement plan may help. One of the big problems with architecture improvement is that customers don't care about architecture. They care about features, reliability, delivery dates, and cost, which makes it hard to spend development dollars on architecture.

To incrementally improve software architecture we need to anticipate future requirements and build in flexibility points for these future requirement changes. Using the open/closed principle,* we can add architectural elements to the system to support the expected requirement changes better, thereby incrementally improving the software architecture. We have to pick our improvement targets carefully. We have hindsight to show the kinds of changes that have occurred over and over during the life of the product. We have marketing predictions that help us anticipate future changes. With this information, an architectural improvement vision can be established. Improvements are prioritized, and some improvements are made with the next release. The newly captured architecture must be documented and publicized to the rest of the development team, so that its existence is known for future release. With each release, more architecture is captured, incrementally improving the architecture.

James Grenning is President of High Point Software Technology Company in Hawthorn Woods, IL. He can be contacted at jgrenning@HighPointSoftware.com.

* According to Bertrand Meyer's *Object-Oriented Software Construction*, the open/closed principle states that modules should be open for extension but closed for modification. Specifically, we can design interfaces that insulate our client applications from embedded services, allowing application extensions by adding new services to the system that obey the defined interface.

References

1. Rising, L. *The Pattern Almanac*, Addison-Wesley, Reading, MA, 2000.
2. http://c2.com/ppr.
3. http://hillside.net/patterns/Lists.html.

This article originally appeared in the
December 1999 issue of *Journal of C++ Report*.