The C++ Style Sweet Spot

A Conversation with Bjarne Stroustrup, Part I

by Bill Venners

Bjarne Stroustrup is the designer and original implementer of C++. He is the author of numerous papers and several books, including *The C++ Programming Language* (Addison-Wesley, 1985–2000) and *The Design and Evolution of* C++ (Addison-Wesley, 1994). He took an active role in the creation of the ANSI/ISO standard for C++ and continues to work on the maintenance and revision of that standard. He is currently the College of Engineering Chair in Computer Science Professor at Texas A&M University.

On September 22, 2003, Bill Venners met with Bjarne Stroustrup at the JAOO conference in Aarhus, Denmark. In this interview, which will be published in multiple installments on Artima.com, Stroustrup gives insights into C++ best practice. In this first installment, Stroustrup describes how C++ programmers can reconsider their style of C++ use to gain maximum benefit from the language.

Climbing above C-level

Bill Venners: In an interview, you said, "The C++ community has yet to internalize the facilities offered by standard C++. By reconsidering the style of C++ use, major improvements in ease of writing, correctness, maintainability, and efficiency can be obtained." How should C++ programmers reconsider their style of C++ use?

Bjarne Stroustrup: It's always easier to say what not to do, rather than what to do, so I'll start that way. A lot of people see C++ as C with a few bits and pieces added. They write code with a lot of arrays and pointers. They tend to use `new` the way they used `malloc`. Basically, the abstraction level is low. Writing C-style code is one way to get into C++, but it's not using C++ really well.

I think a better way of approaching C++ is to use some of the standard library facilities. For example, use a `vector` rather than an array. A `vector` knows its size. An array does not. You can extend a `vector`'s size implicitly or explicitly. To get an array of a different size, you must explicity deal with memory using `realloc`, `malloc`, `memcpy`, etc. Also, use inline functions rather than macros, so you don't get into the macro problems. Use a C++ `string` class rather than manipulating C strings directly. And if you've got a lot of casts in the code, there's something wrong. You have dropped from the level of types, a high level of abstraction, down to a level of bits and bytes. You shouldn't do that very often.

To get out of writing low level code, you needn't start writing a lot of classes. Instead, start using facilities provided in libraries. The standard library os the first and most obvious source, but there are also good libraries for things like math or systems programming. You don't have to do threading at the C level. You can use a C++ threading library. There are quite a few of them. If you want callbacks, don't use just plain C functions. Get libsigc++, and you'll have a proper library that deals with callbacks—callback classes, slots and signals, that kind of stuff. It's available. It's conceptually closer to what you're thinking about anyway. And you don't have to mess with error prone details.

Most of these techniques are criticized unfairly for being inefficient. The assumption is that if it is elegant, if it is higher level, it must be slow. It could be slow in a few cases, so deal with those few cases at the lower level, but start at a higher level. In some cases, you simply don't have the overhead. For example, `vector`s really are as fast as arrays.

Object-Orientaphilia

The other way people get into trouble is exactly the opposite. They believe that C++ should be an extremely high level language, and everything should be object-oriented. They believe that you should do everything by creating a class as part of a class hierarchy with lots of virtual functions. This is the kind of thinking that's reflected in a language like Java for instance, but a lot of things don't fit into class hierarchies. An integer shouldn't be part of a class hierarchy. It doesn't need to. It costs you to put it there. And it's very hard to do elegantly.

You can program with a lot of free-standing classes. If I want a complex number, I write a complex number. It doesn't have any virtual functions. It's not meant for derivation. You should use inheritance only when a class hierarchy makes sense from the point of view of your application, from your requirements. For a lot of graphics classes it makes perfect sense. The oldest example in the book is the shape example, which I borrowed from Simula. It makes sense to have a hierarchy of shapes or a hierarchy of windows, things like that. But for many other things you shouldn't plan for a hierarchy, because you're not going to need one.

So you can start with much simpler abstractions. Again the standard library can provide some examples: `vector`, `string`, complex number. Don't go to hierarchies until you need them. Again, one indication that you've gone too far with class hierarchies is you have to write casts all the time, casting from base classes to derived classes. In really old C++, you would do it with a C style cast, which is unsafe. In more modern C++, you use a dynamic cast, which at least is safe. But still better design usually leads you to use casting only when you get objects in from outside your program. If you get an object through input, you may not know what it is until a bit later, and then you have to cast it to the right type.

Bill Venners: What is the cost of going down either of those two paths, being to low-level or too enamored with object-orientation? What's the problem?

Bjarne Stroustrup: The problem with the C way is that if you write code C-style, you get C-style problems. You will get buffer overflows. You will get pointer problems. And you will get hard to maintain code, because you're working at a very low level. So the cost is in development time and maintenance time.

Going to the big class hierarchy is again, you write more code than you need to, and you get too much connection between different parts. I particularly dislike classes with a lot of get and set functions. That is often an indication that it shouldn't have been a class in the first place. It's just a data structure. And if it really is a data structure, make it a data structure.

Classes should enforce invariants

Bjarne Stroustrup: My rule of thumb is that you should have a real class with an interface and a hidden representation if and only if you can consider an invariant for the class.

Bill Venners: What do you mean by invariant?

Bjarne Stroustrup: What is it that makes the object a valid object? An invariant allows you to say when the object's representation when it's good and when it isn't. Take a `vector` as a very simple example. A `vector` knows that it has n elements. It has a pointer to n elements. The invariant is exactly that: the pointer points to something, and that something can hold n elements. If it holds n+1 or n-1 elements, that's a bug. If that pointer is zero, it's a bug, because it doesn't point to anything. That means it's a violation of an invariant. So you have to be able to state which objects make sense. Which are good and which are bad. And you can write the interfaces so that they maintain that invariant. That's one way of keeping track that your member functions are reasonable. It's also a way of keeping track of which operations need to be member functions. Operations that don't need to mess with the representation are better done outside the class. So that you get a clean, small interface that you can understand and maintain.

Bill Venners: So the invariant justifies the existence of a class, because the class takes the responsibility for maintaining the invariant.

Bjarne Stroustrup: That's right.

Bill Venners: The invariant is a relationship between different pieces of data in the class.

Bjarne Stroustrup: Yes. If every data can have any value, then it doesn't make much sense to have a class. Take a single data structure that has a name and an address. Any string is a good name, and any string is a good address. If that's what it is, it's a structure. Just call it a `struct`. Don't have anything private. Don't do anything silly like having a hidden name and address field with `get_name` and `set_address` and `get_name` and `set_name` functions. Or even worse, make a virtual base class with virtual `get_name` and `set_name` functions, and override it with the one and only representation. That's just elaboration. It's not necessary.

Bill Venners: It's not necessary because there's one and only representation. The justification is usually that if you make it a function, then you can change the representation.

Bjarne Stroustrup: Exactly, but some representations you don't change. You don't change the representation of an integer very often, of a point, of a complex number. You have to make design decisions somewhere.

And the next stage, where you go from the plain data structure to a real class with real class objects, could be that name and address again. You probably wouldn't call it `name_and_address`. You'll maybe call it `personnel_record` or `mailing_address`. At that stage you believe name and address are not just strings. Maybe you break the name down into first, middle, and last name strings. Or you decide the semantics should be that the one string you store really has first, middle, and last name as parts of it. You can also decide that the address really has to be a valid address. Either you validate the string, or you break the string up into first address field, second address field, city, state, country, zip code, that kind of stuff.

When you start breaking it down like that, you get into the possibilities of different representations. You can start deciding, does it really add to have private data, to have a hierarchy? Do you want a plain class with one representation to deal with, or do you want to provide an abstract interface so you can represent things in different ways? But you have to make those design decisions. You don't just randomly spew classes and functions around. And you have to have some semantics that you are defending before you start having private data.

The way the whole thing is conceived is that the constructor establishes the environment for the member functions to operate in, in other words, the constructor establishes the invariant. And since to establish the invariant you often have to acquire resources, you have the destructor to pull down the operating environment and release any resources required. Those resources can be memory, files, locks, sockets, you name it—anything that you have to get and put back afterwards.

Designing simple interfaces

Bill Venners: You said that the invariant helps you decide what goes into the interface. Could you elaborate on how? Let me attempt to restate what you said, and see if I understand it. The functions that are taking any responsibility for maintaining the invariant should be in the class.

Bjarne Stroustrup: Yes.

Bill Venners: Anything that's just using the data, but not defending the invariant, doesn't need to be in the class.

Bjarne Stroustrup: Let me give an example. There are some operations you really can't do without having direct access to the representation. If you have an operation that changes the size of a `vector`, then you'd better be able to make changes to the number of elements stored. You move the elements and change the size variable. If you've just got to read the size variable, well, there must be a member function for that. But there are other functions that can be built on top of existing functions. For example, given efficient element access a find function for searching in a vector is best provided as a non-member.

Another example would be a `Date` class, where the operations that actually change the day, month, and year have to be members. But the function that finds the next weekday, or the next Sunday, can be put on top of it. I have seen `Date` classes with 60 or 70 operations, because they built everything in. Things like `find_next_Sunday`. Functions like that don't logically belong in the class. If you build them in, they can touch the data. That means if you want to change the data layout, you have to review 60 functions, and make changes in 60 places.

Instead, if you build a relatively simple interface to a `Date` class, you might have five or ten member functions that are there because they are logically necessary, or for performance reasons. It's hard for me to imagine a performance reason for a `Date`, but in general that's an important concern. Then you get these five or ten operations, and you can build the other 50 in a supporting library. That way of thinking is fairly well accepted these days. Even in Java, you have the containers and then the supporting library of static methods.

I've been preaching this song for the better part of 20 years. But people got very keen on putting everything in classes and hierarchies. I've seen the `Date` problem solved by having a base class `Date` with some operations on it and the data protected, with utility functions provided by deriving a new class and adding the utility functions. You get really messy systems like that, and there's no reason for having the utility functions in derived classes. You want the utility functions to the side so you can combine them freely. How else to I get your utility functions and my utility functions also? The utility functions you wrote are independent from the ones I wrote, and so they should be independent in the code. If I derive from class `Date`, and you derive from class `Date`, a third person won't be able to easily use both of our utility functions, because we have built dependencies in that didn't need to be there. So you can overdo this class hierarchy stuff.