Abstraction, libraries, and efficiency in C++

Bjarne Stroustrup

bs@cs.tamu.edu

Texas A&M University

Typically, the aim of a responsible programmer is to reasonably quickly produce a program that correctly performs its task, does so with reasonable efficiency, and is maintainable. The key to fast development, correctness, efficiency, and maintainability is to use a suitable level of abstraction supported by good libraries. This short paper provides a view of how C++ can be used to support that aim. The aim of this is not to be "advanced", but to make a couple of simple but important points.

Doing anything significant using only basic language features – that is, without the use of libraries – is unpleasant and unproductive. The key to elegant code and productivity lies in the production and use of libraries. That's true in any language. C++ provides features, such as classes, class hierarchies, templates, exceptions, and namespaces, to directly support library building and use [Stroustrup,2000]. The design of C++ favors systems programming and applications where performance is essential or where resources must be carefully managed [Stroustrup,1994].

Consider a very simple example: To read an unknown number of characters from a keyboard, file, or network connection, we need to set aside a fixed size buffer, read individual characters into that buffer, check that we are not overflowing our allocation (or we'll face memory corruptions and could cause security problems), expand the buffer as needed (or truncate the input), and finally report the number of characters read back to the application. Doing all this "by hand" is tedious and error prone. The C++ standard library provides types, such as **string** and **istream** (input stream) that makes such operations trivial to write. For example:

```
string s;
in >> s;// "in" is an input stream connected to a data source
cout << "I read " << s.length() << "characters";
```

A library provides a vocabulary for saying things directly and simply, correctly handling difficult low-level details in the implementation. For an **istream**, >> means read into; for an **ostream,** << means write to.

Typically, we want to read data with more complicated structure and store it in a form suitable for processing. For example, you might find code like this in software used for managing a movie theatre:

```
typedef map<string, vector<double> > Sales; // a data structure for recording sales

Sales sales      // sales of named movies
string movie;    // name of movie
double take;     // sales amount for movie
while (cin>>movie>>take)       // read name and sales amount
        sales[movie].push_back(take); // add sales amount to movie record
```

This reads sales figures like this

```
Casablanca 123.75
Tom-Jones 234.00
Casablanca 79.45
```

and produces a **map**, that is a data structure that store elements so they can be accessed by indexing with a name string (usually called a key) or by traversing the structure. Typically, a **map** is implemented by a red-black tree [Sedgewick,1998]. Here each element in the **map** is a **string** with a **vector** of its associated sales figures. You could illustrate the structure of resulting sales map like this:

(Casablanca, (123.75, 79.45)) (Tom-Jones, (234.00))

Given that, you can easily and efficiently compute all kinds of useful information, such as total sales for a movie, a list or all movies in alphabetical order, average sales for a set of movies. For example:

```
void print_totals(const Sales& s)
{
        for (Sales::const_iterator p = s.begin(); s!=s.end(); ++p)
                cout << p->first << ": "
                   << accumulate(p->second.begin(),p->second.end(),0.0) << '\n';
}
```

The for-statement visits each element of the **map** in order. The **accumulate** function is a standard library algorithm adding the elements of a sequence. A call looks like this:

```
print_totals(sales);
```

For the trivial input mentioned above, the output would be:

```
Casablanca: 203.20
Tom-Jones: 234.00
```

Maybe it's unrealistic to use a file title as a key when we could use a simple unique integer instead. That would require minimal change to the code:

```
typedef map<int, vector<double> > Sales;
```

Only code that specifically relies on the key being a string would require further modification. In particular, the code fragment reading input and **print_totals()** remains unchanged.

The point of this simple example is to demonstrate to the flexibility and power of C++'s type system. When used in a well-designed library, such as the C++ standard library, the results can be simultaneously elegant and efficient. By elegance, I mean the ability to say things directly without lots of spurious notation. This is a major component of maintainability. Once a type is known, appropriate operations can be applied and appropriate notation defined. It is essential that a user – in particular a user writing a library – can define such types and operations on them. That's the basis of data abstraction, object-oriented programming, and generic programming.

C++ supports each of these programming styles, sometimes called "programming paradigms" and allows a programmer to choose between them to best express solutions to various problems. This is often called "multi-paradigm programming" and rests on the observation that many problems have more elegant or efficient solutions using one style than another, that no one style is superior in all cases, and that some problems are best solved using a combination of techniques. In particular, library writers benefit from the efficiency and flexibility offered my multi-paradigm programming.

In the example, I used simple generic programming as supported by the standard library. For example, the **map** and **vector** types are parameterized on their element types offer a variety of common useful operations, such as **accumulate**() and the iteration mechanism used. The elegance of such code comes from the compiler's ability to automatically choose the correct implementation of operations based on the declared type of each object. I did not have to litter my code with complicated names or inelegant type conversions to correctly select appropriate operations for each type of object. To compare, try rewriting the example in some other language or in some other style of C++.

Like elegance, efficiency comes from knowing the type of each object at compile time. For a small example like the one above, performance is probably not an issue. However, C++ is used in application areas where performance is essential, such as high-energy physics, gaming, and some embedded system applications [Stroustrup,2003]. In such applications, programs usually rely on highly tuned libraries providing fundamental operations. For example, there are vector and matrix libraries that make the performance of standard scientific and numeric computations as good as that of Fortran while significantly improving notation. For example, given a suitable library (such as Blitz++ [Veldhuizen,2003], MTL [Siek,1999], uBLAS [Walter,2003], or POOMA [Haney,1999]), you can write code such as this without loss of performance compared to Fortran:

```
Matrix<double,100,50> m;
Vector<double,100> v,v2;
// initialize m, v and v2
Vector<double,100> v3 = m*v+v2;
```

Furthermore, such libraries can express more advanced (or more specialized) concepts, such as:

```
Matrix<int,sparse,10000,500> m2;
Matrix<complex<double>,UpperTriangular,100,50> m3;
```

The key to performance is that the declared types are used to guide the selection of implementations. The techniques used are standard part of C++ generic programming. A more detailed discussion of C++ performance issues can be found in the ISO C++ standard committee's recent Performance technical Report [Goldthwaite,2003].

In addition to being key to ease of use, correctness, and performance, libraries are key to portability. Like C, C++ contains many constructs that are implementation defined; that is, they detailed meaning varies from platform to platform. This is essential for low-level work (such as direct hardware access) and often essential for performance at the lowest level. However, if a programmer directly relies on such a feature, the result is clearly a non-portable program. The obvious solution for most programmers is to use a library that provides an elegant and efficient platform independent to needed services. There are many such libraries. Examples are BOOST [BOOST,2003] and ACE [Schmidt,2002]. Using such libraries, a C++ programmer can use a range of systems unmatched by any other language except C, while retaining whichever degree of platform independence is desired.

To actually get work done, you need more than a language specification. For example, you need a compiler that works for your particular computer and operating system, libraries, manuals, tutorial material, and a technical community. This is one of C++'s strengths. C++ is supported by all major platform vendors and implemented according to its international (ISO) standard [Koenig,2003]. It is therefore less vulnerable to changes of fashion and the whims of commercial organizations than proprietary languages.

How do you decide if the strengths of C++ are applicable to your particular problems? You basically have to listen to people who have tried it and try for yourself. In doing so, it is most important to seek out good libraries and use them, rather than making the – unfortunately common – mistake of trying to write everything yourself from the bare language features or using only a few functions remembered from the C standard library. In particular, for most people it is essential to raise the level of programming style from the tradition C and Fortran styles to a more abstract, yet still efficient, level relying on libraries. If no other library is relevant, the C++ standard library can often be used. For example, see my comparison of C-styles and C++ styles in "Learning Standard C++ as a New Language" [Stroustrup,1999].

References

[BOOST,2003] http://www.boost.org.

[Goldthwaite,2003] Lois Goldthwaite (editor): "Technical Report on C++ Performance". SC22/WG21. Link on http://www.research.att/~bs/C++.html.

[Haney,1999] Scott Haney and James Crotinger: "How templates enable high-performance computing in C++". IEEE Computing in Science and Engineering. 1999. http://www.acl.lanl.gov/pooma/papers.html.

[Koenig,2003] Andrew Koenig (editor): "International Standard for the C++ Programming Language". ISO/IEC 14882:2003(E). Link on http://www.research.att/~bs/C++.html.

[Schmidt,2002] D. C. Schmidt and S. D. Huston: "C++ Network programming". Addison-Wesley. 2002. http://www.cs.wustl.edu/~schmidt/ACE.html.

[Sedgewick,1998] Robert Sedgewick: "Algorithms in C++ (3rd edition)". Section 13.4. Addison-Wesley. 1998.

[Siek,1999] Jeremy Siek and Andrew Lumsdaine: "The Matrix template Library: Generic components for High-performance Scientific Computing". IEEE Computing in Science and Engineering. Nov/Dec. 1999. http://www.osl.io.edu/research/mtl.

[Stroustrup,1994] Bjarne Stroustrup: "The Design and Evolution of C++". Addison-Wesley. 1994.

[Stroustrup,1999] Bjarne Stroustrup: "Learning Standard C++ as a New Language". C/C++ Users Journal. May 1999. Link on http://www.research.att.com/~bs/papers.html.

[Stroustrup,2000] Bjarne Stroustrup: "The C++ Programming Language (Special Edition)". Addison-Wesley. 2000.

[Stroustrup,2003] http://www.research.att.com/~bs/applications.html.

[Veldhuizen,2003] Tod Veldhuizen: "Blitz++". http://www.oonumerics.org/blitz.

[Walter,2003] Joerg Walter and Mathias Koch: "uBLAS – Basic Linear Algebra". http://boost.org/libs/numerics/ublas/doc/index.htm.