C++ Programming Styles and Libraries

Bjarne Stroustrup

AT&T Labs - Research
Florham Park - NJ

ABSTRACT

One of the main aims of C++ has been to make it an excellent tool for writing libraries. Here, I present some points about the role of libraries and of the programming styles that they support and rely on. For lack of space for a thorough treatment of these themes, I refer to books.

1 Introduction

C++ is a multiparadigm programming language. That is, C++ supports several styles of programming:

C-style programming	C++ is a better C, maintaining C's flexibility and run-time efficiency while improving type checking;
Data abstraction	the ability to create types that suit our needs;
Object-oriented programming	the idea of programming with class hierarchies and runtime polymorphism; and
Generic programming	programming using type parameterization of both data types and algorithms.

Importantly, C++ supports the use of combinations of those styles. This is crucial because the most effective programming techniques involve a variety of styles that people often classify as different. Radically different programming styles are often referred to as different paradigms; hence the word ‘‘multiparadigm’’ [Stroustrup,2001].

Naturally, this flexibility is viewed as mere complexity by people who think that there is one style of programming that is right for everyone. However, C++ is a general-purpose programming language with a bias towards systems programming and none of the candidates for ‘‘the one right way’’ of writing programs adequately supports the range of needs faced by C++ programmers.

One group of programmers in particular need generality, flexibility, and efficiency beyond what most programmers consider normal and reasonable: library writers. Many parts of C++ are best understood as facilities supporting library writers and library users. Think of it this way: Without a good library, most interesting tasks are hard to do in C++; but given a good library, almost any task can be made easy. Here ‘‘easy’’ means that the programming language isn't a source of significant complexity so that a programmer can concentrate on the fundamental problems of the task in hand.

2 An example

Here is an example illustrating the power of libraries:

```
void Decompose(double delt, SymTensor& V, Tensor& R, const Tensor& L)
{
    Symtensor D = Sym(L);
    AntiTensor W = Anti(L);
    Vector z = Dual(V*D);
    Vector omega = Dual(W) - 2.0*Inverse(V-Tr(V)*One)*z;
    AntiTensor Omega = 0.5*Dual(omega);
    R = Inverse(One-0.5*delt*Omega) * (One+0.5*delt*Omega)*R;
    V += delt*Sym(L*V-V*Omega);
}
```

According to [Budge,1992], ‘‘This code is transparent and ... a physicist familiar with the polar decomposition algorithm can make immediate sense of this code fragment without the need for additional documentation.’’

I’m not a physicist, so that code is not obvious to me, but it is clear how that code is trivially translated from what you find in a physics textbook. This example is taken from a pioneering library from Sandia Labs. However, such libraries are now common in various branches of high-energy physics and general vector and matrix libraries supporting them are widely used and available.

I chose physics as my example partly because it is a field where performance is essential. If a program in this field is significantly slower than a conventional Fortran program, it will be discarded. In this area, elegance of programs traditionally takes the back seat compared to performance. The reason is simple: this is a field where results are often limited by available computing resources. C++ and modern programming techniques have met that challenge. Using generic programming techniques, C++ vector and matrix libraries have equaled and exceeded the performance of classical Fortran code (for example, POOMA [POOMA] MTL [MTL] and ROOT [ROOT]). One key to this exceptional performance is the elimination of temporaries through the use of small function objects [Stroustrup,2000§22.4.7]. Another key is (other) function objects representing critical implementation policies. Basically, the average programmer cannot write code that equals the modern libraries in numerical performance. Good programmers often know better than to even try.

Different fields have different requirements. This leads to the use of different programming styles being key to the design and use of different libraries. Numeric libraries tend to rely on generic programming emphasizing conventional notation and efficiency. To contrast, libraries focusing on graphics or user interaction tend to rely heavily on object-oriented techniques involving class hierarchies. When number crunching meets data management and graphics, combinations of programming techniques occur naturally.

The time taken to write a program is at best roughly proportional to the number of lines written, and so is the number of errors in that code. If follows that a good way of writing correct programs is to write short programs. In other words, we need good libraries to allow us to write correct code that performs well. This in turn means that we need libraries to get our programs finished in a reasonable time. In many fields, such C++ libraries exist. The standard library is an excellent way to start getting away from the messier and more error-prone levels of programming [Stroustrup,2000] [Stroustrup,1999]. However, there is much more to C++ than the standard library – that library is just an essential beginning.

3 Education and Libraries

The best an individual or an organization can do to improve their ability to produce good code (that is clean, maintainable, efficient code) is to master modern C++ programming styles and use good libraries. The best best an individual or an organization can do for the general community is to contribute a good, well-documented library for a field where adequate libraries don’t yet exist.

In this context, I’d like to mention some books. These are books on programming style and libraries that I have had a small hand in as an editor. In several cases, I have encouraged the writing of these books because I thought their topic important. In every case, I have nagged the author to try to make the book shorter and more accessible to the reader.

‘‘Accelerated C++’’ [Koenig,2000] is an innovative textbook that teaches Standard C++ from scratch to beginners assumed to have programmed before in some other language. Its presents every feature in a context where the feature is realistically used effectively, relies on the standard library to teach effective programming styles, and doesn’t introduce basic language facilities (such as pointers and arrays) until they are needed. It is also a most useful book for C++ programmers who wants to move on to more modern styles of C++ programming. For example, ‘‘Accelerated C++’’ introduces templates several chapters before pointers.

‘‘Exceptional C++’’ [Sutter,2000] explores the uses of exceptions in quite some depth. Similarly, ‘‘Modern C++ Design’’ [Alexandrescu,2001] assumes that a reader understands the basis of templates and generic programming proceeds to demonstrate several advanced and interesting techniques. These are books for people who are not satisfied with introductory-level explanations or simple descriptions of language facilities. They expand into advanced programming techniques and into technical details that we – most of the time – prefer to remain hidden inside libraries. For most C++ programmers, these books combine the intellectual thrill of discovering something new with practical techniques for building and using libraries.

‘‘Modern C++ Design’’serves as a bridge to the second type of books that I want to mention: The documentation of the design and facilities of interesting C++ libraries. ‘‘Modern C++ Design’’ presents Alexandrescu’s ‘‘Loki’’ library for building generic components, many based on classical patterns.

‘‘C++ Network Programming’’ [Schmidt,2002] describes ACE, a very large and very successful library for programming distributed systems. Its roots go back to the early 1990s, and it is in serious industrial use. Much of what people think of doing with threads, sockets, locks, etc., is already being done using ACE. My favorite ACE story happened when a developer was explaining a large industrial system for medical applications at a conference. He emphasized his needs for portability and someone asked the inevitable question: ‘‘so why didn’t you use Java?’’ I had expected one of the usual answers (stressing performance, flexibility, or elegance), but he simply responded: ‘‘because ACE runs on more platforms than Java’’ and carried on with his talk. The ACE style combines classical object-oriented programming with extensive use of templates for type safety and flexibility.

The final book that I’ll mention here, ‘‘The Boost Graph Library’’ [Siek,2002] describes a new library. Its authors contributed the MTL (Matrix Template Library [MTL]) to the numerical computation community and then proceeded to apply the lessons learned to the area of graph algorithms. The result, the BGGL (Boost Generic Graph Library) is a tour de force of generic programming. There is a heavy emphasis on flexibility and the book also reflects many of the lessons learned about how to document generic libraries. It is an example of what you can achieve when you take a fresh look at an area applying the tools of Standard C++ [ISO,1998]. ‘‘Boost’’ is a collection of open source libraries designed to augment the C++ standard library [Stroustrup,2000] and a collection of people writing and documenting them [BOOST].

The main point of this article has been to encourage people to build and use good libraries by giving examples and a few simple arguments. Understanding the needs of a potential user community and matching those with design styles and programming techniques is harder than most programmers are willing to believe. Designing, implementing, and documenting a library for an interesting application domain is a worthy intellectual challenge. Getting a library into widespread use within its intended user community is another major challenge involving teaching at many levels. To succeed as users, designers, and implementers, we need to study examples of what can be done and what has been done. Good luck and have fun!

4 References

[Alexandrescu,2001]	A. Alexandrescu: *Modern C++ Design*. Addison-Wesley. ISBN 0-201-70431-5. 2001.
[BOOST]	A growing collection of open source libraries designed to augment the C++ standard library. http://www.boost.org.
[Budge,1992]	Ken Budge, J.S. Perry, and A.C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
[ISO,1998]	ISO/IEC 14882, Standard for the C++ Language.
[Koenig,2000]	Koenig and B. Moo: *Accelerated C++*. Addison-Wesley. ISBN 0-201-70353-X. 2000.
[MTL]	MTL (Matrix Template Library) from University of Notre Dame. http://www.osl.iu.edu/research/mtl/.
[POOMA]	POOMA (Parallel Object-Oriented Methods and Applications) from Los Alamos National Labs. http://www.acl.lanl.gov/Pooma/. This library is used for parallel scientific computation.

[ROOT] ROOT (from CERN. http://root.cern.ch/root/. This library is used for data analysis of large volumes (for example, many terrabytes) of experimental data.

[Schmidt,2002] D. C. Schmidt and S. D. Huston: *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley. ISBN 0-201-60464-7. 2002.

[Siek,2002] J. Siek, L. Lee, and A. Lumsdaine: *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley. ISBN 0-201-72914-8. 2002.

[Stroustrup,1999] B. Stroustrup: *Learning Standard C++ as a New Language*. C/C++ Users Journal. pp 43-54. May 1999. Also in CVU Vol 12 No 1. January 2000. http://www.research.att.com/~bs/new_learning.pdf.

[Stroustrup,2000] B. Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. ISBN 0-201-70073-5. 2000.

[Stroustrup,2001] B. Stroustrup: *C++ glossary*. http://www.research.att.com/~bs/glossary.html

[Sutter,2000] H. Sutter: *Exceptional C++*. Addison-Wesley. ISBN 0-201-61562-2. 2000.