Joslyn, Cliff A: (1993) "Review: C++ Programming Style, by Tom Cargill", Computing Reviews, v. 34:4, pp. 192-193

Review of *C++ Programming Style*, by Tom Cargill

Cliff Joslyn * †

September, 1992

Perhaps this helpful little book is mistitled. The term "programming style" can connote issues of purely *lexical* style, that is code formatting (for example, indentation methods) and coding standards (for example, variable naming conventions). A more descriptive title might be *C++ Programming Practice*. Cargill, a well-known author on C++ programming techniques, quite explicitly avoidis issues of lexical style, instead concentrating on the gray area between style and design, and on what is important for both general programming methods, for any object-oriented (OO) languages, and for C++ in particular. Cargill reveals many of the effective techniques and knowledge of the likely problems that have been accumulated, but rarely documented, by the C++ programming community.

This book will be very useful and welcome to any serious C++ programmer who has learned C++ syntax, but realizes that much more knowledge is still required before becoming *competent* in C++. Only a few other works exist that attempt to achieve the same goals for either C or C++. Some of those are published [3, 5], while others are available only over the Internet [1, 2].

The chosen title is a direct reference to Kernighan and Plauger's classic *The Elements of Programming Style* [4]. Cargill reflects the earlier work by presenting a series of principles in eight chapters concerning abstraction, consistency, single inheritance, virtual functions, operator overloading, C language wrappers, efficiency issues, and multiple inheritance, with an extra chapter combining much of the above in a small application. Each chapter is devoted to the analysis and rewriting of a poorly designed program. These programs were previously published, and since they come from different (unidentified) authors, the reader is exposed to a variety of different lexical styles. Each chapter concludes with a summary, appropriate references, and an exercise or two.

Some of the principles which this code reworking illustrates echo the original principles of Kernighan and Plauger:

- Replace repetitive expressions by calls to a common function.
- Write clearly — don't be too clever.

Others are generic to any OO language (although described in C++ terminology):

- A class should describe a set of objects.
- Use data members for variation in value; reserve virtual functions for variation in behavior.
- Independent objects should have independent behavior.
- Reduce coupling — minimize interactions between classes.

*Graduate Fellow, Systems Science, SUNY–Binghamton, 327 Spring St. # 2, Portland ME, 04102, USA, (207) 774-0029, cjoslyn@bingsuns.cc.binghamton.edu, joslyn@kong.gsfc.nasa.gov.

†Supported under NASA Grant # NGT 50757.

- A constructor should put its object in a well-defined state.
- No class is perfect; too narrow a design is better than too broad.
- Do not encapsulate essential information — make it available by some means.

Some principles are common to both C and C++:

- Remember the null byte — use `new char[strlen(s)+1]`.
- Know the valid lifetime of a pointer returned from a function.

While others are specific to C++, and are very important for effective, error-free C++ code:

- When defining `operator=`, remember `x=x`.
- Consider default arguments as an alternative to function overloading.
- Identify the `delete` for every `new`.

Along the way, many of the techniques and problems that have become common knowledge in the C++ community are introduced, including not only the principles mentioned above, but also the use of reference counts and stream I/O, the need for virtual base constructors, the pitfalls of fixed operator precedence, methods to handle constructor failure, the correct return type for `operator=`, the use of "module" classes with static members, the relation between `operator+=` and `operator+`, operator constructors, efficiency issues related to temporaries, the use of shallow copies under multiple inheritance, the sources of memory leaks, and even the limitations of some existing compilers.

Example programs include simple string and stack classes. Solid OO design principles are emphasized throughout, for example the concept of class invariants; the needs for consistency and completeness; and the distinctions between interface and implementation, state and behavior, and logical and physical state.

Templates and exceptions are not treated, except in passing. We can only hope that a revision of *C++ Programming Style* that addresses these methods will be available once the C++ community has had enough experience with them to provide useful guidance.

References

[1] Cannon, L.W.; Ellliott, R.A.; et al. *Recommended C Styles and Coding Standards*, "Indian Hill" style guide, available by anonymous `ftp` from `cs.washington.edu` in `~ftp/pub/cstyle`, 1991.

[2] Cline, M. *Answers to Frequently Asked Questions on* `comp.lang.c++`, available by anonymous `ftp` from `sun.soe.clarkson.edu` in `~ftp/pub/C++`, 1992.

[3] Coplien, J.O. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading MA, 1992.

[4] Kernighan, B.W; and Plauger, P.J. *Elements of Programming Style*, McGraw Hill, New York, 1974.

[5] Koenig, A. *C Traps and Pitfalls*, Addison-Wesley, Reading, MA, 1989.