General Coding Standards

Ken Clowes (kclowes@ee.ryerson.ca)

November 11, 2000

Contents

1 Basic principles

Developing software requires some discipline. One component of this discipline that is often underestimated by beginners is the adherence to some set of general principles for organizing and writing source code. Many of these principles are are sufficiently general that they should be applied to all software irrespective of the programming language. For example, the principles described here can be applied to C, Assembler, Matlab, Java, C++, VHDL, Maple, tcl, Perl, shell scripts and others.

The source code for programming projects should always be organized and written with the future tasks of testing, debugging and maintenance (possibly by others) in mind. The coding standards described here deal with the larger issues that make these tasks easier. The most fundamental principle is *consistency*—whatever detailed coding standards you adopt, you should always follow them consistently.

The rest of this document expands on the basic principles outlined below:

Public documentation: Source code should contain specially marked comments that provide sufficient knowledge for a reader to use the software product without examining the actual implementation.

Private documentation: Private comments explain details of the implementation. Only someone wishing to modify or understand the implementation would read these comments.

Naming conventions: Sensible and consistent names should be used.

Organization: The directory hierarchy and standard files (like `README` files) should be organized clearly and consistently.

Formatting: Visual clues should be used to indicate the logical organization of the source code.

2 Public Documentation

Public documentation explains how to use software—i.e. it describes the *interface* between the software product and its users; it does *not* describe implementation details. We consider two general kinds of software and users:

Programs: One kind of product is a runnable program that anyone can use. For example, a utility program can be written in a scripting language or C (with a `main` function), etc. A user of such program should need no knowledge of the implementation language; they only need to know how to invoke the program by reading the "man page" entry.

In these cases, the public comments could, for example, be specially designed so that a "man page" or HTML description could be generated automatically from the source code public comments.

Libraries: The other kind of software product is a library of useful functions, subroutines or classes that a programmer of the implementation language could exploit. In this case the user of the software product is assumed to be skilled in the source code language.

For example, the source code for a package of Java classes should contain public comments that can be used to automatically generate all the documentation a user of the package needs to exploit its features.

Some other languages such as C, C++ and elisp also have tools or conventions that allow the automatic generation of user documentation.

If the language does not have such tools, the programmer should develop their own standards for clearly identifying comments that are public. For example, one convention that I use for assembly language programming is to start all lines that are public comments with "`;;`".

Public documentation should be accurate, but terse. If more lengthy descriptions are required, a separate user manual or on-line help interface should be designed and their existence should be stated in the public documentation.

It is also recommended that the public documentation of interfaces be written before actually coding the implementation. Specifying the interface focuses the mind on the essentials and allows the designer to review the interface looking for inconsistencies or ambiguities before committing it to code.

3 Private Documentation

Private comments are meant to be explain implementation details of the source code.

Since the reader of these comments is assumed to be competent in the programming language, obvious comments that clutter the code and insult the reader's intelligence should be avoided. For example, code like:

```
i++;  /* Increment i */
```

should be avoided.

Often, no private comments are required at all in well written programs. The use of descriptive names is also a great help. Indeed, Rob Pike states:

> Basically, avoid comments. If your code needs a comment to be understood, it would be better to rewrite it so it's easier to understand.
>
> Rob Pike[Pik]

Avoiding comments completely, however, is sometimes too drastic a measure. The programmer should use common sense and consider the needs of the potential reader. If it is expected that the reader is a seasoned expert in the language, it may be appropriate to have minimal or even no private comments at all. However, if you think the reader may have only a smattering of expertise, more detailed private comments are reasonable. I find this to be especially true for scripting languages. For example, I sometimes write perl scripts but I am not an expert in the language and I insert private comments to remind myself what bizarre language feature I am using. Of course, a perl guru may find such comments superfluous and annoying.

Another situation in which detailed implementation comments are often appropriate is source code that is meant for pedagogical use by people just learning the language.

4 Naming conventions

Names of entities such as functions, variables, classes, etc. should be given meaningful names. As a general principle, the greater the scope of an entity, the more descriptive its name should be. This is especially important for names that are public (i.e. known to the user).

Numeric and string constants should be given symbolic names; this is far preferable than embedding "magic" numbers or messages in the source code.

5 Organization

Projects should be organized in a hierarchical manner. The top-level directory should have at least a `README` file and a `Makefile` (or its equivalent.)

The README file should be a pure text file.

The Makefile's default target should create the project's implementation from source code. Other common targets include:

clean: Removes generated files.

install: Installs the software.

test: Tests the software.

archive: Makes an archive of all the source files.

6 Formatting

You say toe-may-toes
I say toe-mah-toes
Let's call the whole thing off.

—Ira Gershwin (lyrics) and George Gershwin (music)

Source code should be formatted so that visual clues (such as indentation or vertical white space) aid in understanding the organization and logical flow of the implementation.

Different organizations and individuals may have specific detailed conventions that they follow and it is not our intention to stipulate any particular standard (which risks starting religious wars over minor details). However, as a general principle, I do prefer:

- that tabs be replaced with spaces;
- and that lines be limited to 80 characters or less.

These rules ensure that you can always examine the source code on a dumb terminal and that you can embed it in an e-mail message easily.

With the *emacs* editor, there are special editing modes for almost all programming languages. To avoid tabs and limit line length, you can use the `auto-fill` mode and set `indent-tabs-mode` to `nil`.

7 Furthermore...

Excellent advice from master programmers on many aspects of the craft (much more than coding standards) can be found in *The Practice of Programming*[KP99]

I have written more detailed descriptions of coding standards for C[Clob] and assembler[Cloa]

References

[Cloa] Ken Clowes. *Assembly Language Coding Standards.*
`file:~kclowes/public/CodingStds/CodingStdAsm.ps`

[Clob] Ken Clowes. *C Coding Standards.* file: `CodingStdC.ps`.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley, Reading, Massachusetts, 1999. 267 pages.

[Pik] Rob Pike. *Notes on Programming in C.* 5 pages.