The Elements of Style Literate Programming by Kevlin Henney

Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reinforce each other.

Donald E Knuth

When Knuth wrote this he was introducing the rationale underpinning his WEB system for program and document creation [*The Computer Journal* 27(2), 1984], but I believe that as they stand the principles are also applicable to general programming style. The choice of names, comments and layout directly affects the readability and hence the quality of code. It seems that there are as many approaches to these as there are programmers -possibly more, because programmers evolve through various styles. A computer will ultimately be responsible for turning our code into its code, but it is clearly another person or the original author that will be responsible for fixing and extending the system. I do not intend to draw up a single rigid approach to naming and commenting - that is clearly a religious issue - but instead outline a reasonable set of requirements offering a good signal-to-noise ratio and yet not so minimalist as to require telepathy in others. Nonetheless, counterexamples often help to identify what is not reasonable. Consider:

```
/**********************************
 *                                *
 *          Add one to i          *
 *                                *
 **********************************/
                i=i+1;
```

"Don't laugh now, wait until you see it in real life" is Rob Pike's advice regarding this fragment in his *Notes on Programming in C*.

Comments should be clear and concise. We should assume the reader is competent, avoiding restatement of the obvious or lessons in how to use the language. The place for educational comments is in education not in production. On the other hand we should not assume that everyone has the same specialist knowledge; assumptions about the application domain are often worth commenting, briefly.

Over commenting can be just as bad as under commenting. Consider the function definition: this often plays host to very large banners. These often include

a copyright message that should be at the head of a module;

a complete biography of the function which, if necessary, should be placed at the head of the module or handled by a version control system;

a description of what the function does, which should be placed by the function's declaration in a header file and not by its definition - this is interface information, and so it should live in the interface;

a description of the role of all the function's arguments which, if not evident from the name given in the prototype, should also be in the header;

a tag indicating the mode (*in*, *out* and *in-out*) of each argument is an attempt to impose another language model on C and C++ when a prototype already declares the calling mechanism and the const-ness or otherwise of a given argument;

a listing of the return type, argument types and the argument count of a function - as a pointless exercise this one speaks for itself.

Recommendations for this sort of banner comment are often to be found in company coding guidelines and many of us have been guilty of a similar practice at one time or another. Some programmers mistake this style for quality; it is not, it is waffle. Large boiler plates are for modules and not sub-components; programmers are discouraged from writing small functions and classes by such bureaucratic overhead.

Much of the advice on separating interface from implementation in system design is also applicable to commenting [Stroustrup, *The C++ Programming Language*, 2nd edition]:

The ideal interface

- presents a complete and coherent set of concepts to a user,

- is consistent over all parts of a component,

- does not reveal implementation details to a user,

- can be implemented in several ways,

- is statically typed,

- is expressed using application level types, and

- depends in limited and well-defined ways on other interfaces.

Within a function, comments should be limited to what is not and cannot be clearly stated by the code:

portability warnings;

maintenance advice, if not otherwise obvious;

elaboration of anything that is an indirect consequence of a complex design decision or a necessary work around that would otherwise seem obscure;

haxplanation of `clever' or write-only code, although a review of whether this is genuinely required would also be helpful;

and, especially in long functions, section headings for blocks or to break up long sequences.

In the previous *Elements of Style* I mentioned that the *meaning* in *meaningful variable names* is a little nebulous. What forms the basis of a meaningful name? Hopefully conceptual usage, rather than implementation, should play some part. Commenting variable declarations with anything other than the descriptions recommended above is suspicious; put descriptions into names rather than comments:

```
char szUser[L_cuserid] = ""; // user name
char UserName[L_cuserid] = "";
```

Unless used as a warning against the dire consequences of an action, repetition is redundant. Including the comment for the first declaration above after the second would simply restate what is now obvious. In these declarations it is enough to know the purpose of the string. Embedding implementation rather than abstract type information is unnecessary. This is the job of the declaration.

There are some instances where repetition is helpful and necessary for re-emphasis of something that might not otherwise be obvious. Take, for example, redeclaration of C++ virtual member functions as virtual. Overriding a virtual function automatically makes the new function virtual, but this is not always obvious given the size of class declarations and their distribution over many header files:

```
// base header
class Base
{
public:
    virtual void AMethod();
    virtual void AnotherMethod();
};
// derived header
class Derived : public Base
{
public:
    void AMethod(); // not obviously virtual
    virtual void AnotherMethod(); // clearer
};
```

Scope is a rough and ready indicator of identifier length. The greater the scope of a name, the longer it needs to be. Names such as classes, functions and particularly variables should become more specific the more general the context. The one obvious exception to this is argument names, where present, in prototypes. Their scope is to the end of the prototype declaration, but as I mentioned above these carry the weight of interface description with them.

Kernighan and Plauger's telephone test is a useful guideline for identifiers. How easily could you read your program down the phone to be correctly transcribed at the other end? Such guidelines are useful rules of thumb, but they do not imply reviewers walking all over your code with a ruler plotting graphs of identifier length against visibility or reading your listings through to the switchboard!

Check spelling if ever in any doubt: many can grok jargon and Americanisms, but try not to inflict averege, seperator and temparature on the name space. Everyone makes mistakes, but recognising in

advance that this is the case saves more time than having it pointed out. Poor spelling or slow typing is also no excuse for hiding behind obscure abbrs. For instance nothing is gained by spelling Matrix as Mtrx. Beware that eliding vowels and endings in identifiers may reduce code readability or create ambiguity with other domain related words and acronyms. More importantly, it pays to be consistent - consider the following examples from the same well-known system: Win, Wind and Wnd. Uncommon abbreviations are an especially false economy and it will not push anyone over the RSI threshold to make identifiers more telephone friendly.

Lastly, do not go over the top by confusing literate with literary [Bertrand Meyer, *Object-oriented Software Construction*]:

If your readers have literary inclinations, they would rather read Henry James than your comments.