Coding Clearly

This is a computer science department. We’re in the business of training you to be professional programmers. That means a number of things. It means you should know certain standard languages and algorithms; that you should know how to debug; etc. But it also means that you should be able to write clear, comprehensible code. For this reason, all programming assignments in the class will be **code reviewed**. That means that each time there is a programming assignment, you and your fellow students will have to read one another’s code, critique it, and explain it in writing. We’re not doing this to be cruel. We’re doing it because you’ll need to do it when you are working in industry and we want you to start learning now. We’re also doing it because no one really learns to be a good programmer by taking programming classes. They learn to be good programmers by reading other people’s code and getting a sense of what does and does not work.

Although there are no coding standards for this class *per se*, there are a number of common-sense practical techniques for making code easier to read that you should follow lest you incur the rath of your instructor, TA, and fellow students. I’ve codified these as a set of obnoxious aphorisms so that they will be easy to remember. Before turning a program in, you should review these and check whether you have any gross violations of them.

*A program is a form of communication between you and the computer*

The ultimate purpose of a program is to tell the computer how to perform some task. Why state the obvious? To place the following in stark contrast:

*A program is a form of communication between you and a human*

Computer programs are never finished, only abandoned. Except for very simple programs, the last bug will never be fixed, the last feature never integrated. Programs are always being worked on, so they are always being read by humans. Whether that human is your successor at your company or is simply yourself two weeks in the future, they will almost never know how all the code is supposed to work when reading it. A maintainable program must, therefore, be written so as to be comprehensible to someone who doesn’t already understand it. This is arguably the most important aspect of a good program. A fast program isn’t any good to anyone if it’s broken or if no one can figure out enough of it to make it work under the new version of Windows. And as Ken Forbus says, it’s a lot easier to make a clean, slow program fast, than to make a fast, ugly program clean.

*You should write a program in the same way you would write an essay.*

A program should be as concise (read: “short”) as possible so that the reader doesn’t get lost in page after page of verbiage. However, since the whole purpose of conciseness is to make it intelligible, you obviously shouldn’t make the program so short as to sacrifice intelligibility.

You need to think about what the reader can be counted on to understand in advance and what they cannot, taking special care to make the latter clear.

You also need to put some thought and care into the placement of components in the source file(s). Things should be easy to find. Related components should be nearby one another. Modern programming
environments make this somewhat less of an issue (since they can usually bring you to the definition of any particular procedure or variable automatically), but it can still be important, particularly for TAs trying to read your code.

Only the computer reads the program in its entirety

If everyone who worked on Windows 2000 has to read and understand all 40,000,000 lines of code before starting work on a project, Microsoft would never be able to hire any new programmers. So programs need to be written so that you don’t have to read the entire text in order to understand the function and operation of one little section. When writing and commenting code, do not assume that the reader has already read the whole thing.

Computers read the punctuation, humans read the indentation

Consider the C code fragment:

```c
open_window(f, (a++, a->foo)?c:d+7, bla(bleep(x+7, y+2, z++), x+4, z--), c);
```

While this is somewhat complicated, it’s easy to find considerably more complicated code in the business world. And it’s really easy to find uglier code in people’s homework assignments.

Now ask yourself: what is the fourth argument to `open_window`? The answer is “c”, but most people tend to think it’s “x+4”. If you stop and look at it carefully, probably by manually counting parentheses, you’ll get it right. But who wants to have to do that with every passing glance at a fragment of someone’s code? That’s why you should write the code like this:

```c
open_window(f, (a++, a->foo)?c:d+7, bla(bleep(x+7, y+2, z++), x+4, z--), c);
```

Or even:

```c
open_window(f, (a++, a->foo)?c:d+7, bla(bleep(x+7, y+2, z++), x+4, z--), c);
```

Some programmers deal with this by introducing intermediate variables, for example:

```c
arg2 = (a++, a->foo)?c:d+7;
arg3 = bla(bleep(x+7, y+2, z++);
open_window(f, arg2, arg3, c);
```

Personally, I dislike this approach because it means that if I want to know what the second argument to `open_window` is, I have to search backward for the definition of `arg2`. But the whole point of writing clear code is to allow the reader to figure out what they need to know about the code with a minimum of bother. That means they shouldn’t have to do a lot of skimming looking for definitions of things.

Note that if you write it like this:
open_window(f,  
  (a++, a->foo)?c:d+7,  
  bla(bleep(x+7,y+2,z++),  
    x+4,  
    z--),  
  c);

which is indented, but indented wrong, we will be forced to kill you in order to protect society from the buggy code you and your offspring would write if you were allowed to live and procreate. Writing it like this:

open_window(f,  
  (a++, a->foo)?c:d+7,  
  bla(bleep(x+7,y+2,z++),  
    x+4,  
    z--),  
  c);

is considerably less dangerous (and more common), since it makes it obvious to the reader that you are a pinheaded loser whose code should not be trusted, whereas the previous example appears to be competently indented code. However, we would probably have to have you killed anyway.

What I’m trying to say here is that code indentation is not just a matter of making the code look pretty. It’s a matter of making it intelligible to humans. You must indent your code properly for this class. If you do not, it will be returned to you ungraded and counted as late. Or we may just kill you. Or, if we really hate you, we’ll go ahead and pass it out to all your friends for code review and make them figure it out. Then you will have to go into hiding for several weeks until they forgive you.

We have no specific requirements for structuring indentation. However, you must make sure that:

- Given an expression, it’s easy for us to figure out at a glance what its subexpressions are
- Given a subexpression, it’s easy for us to figure out at a glance what its parent expression is

The standard indentation rules for Scheme code are:

- If expression \((E_1 \ E_2 \ldots \ E_n)\) can be kept on one line if none of its subexpressions \(E_i\) have subexpressions themselves. Otherwise, the \(E_i\) should be put on different lines, although \(E_1\) and \(E_2\) are usually kept on the same line. In other words, \((+ \ a \ (- \ c \ d) \ e)\) is cool to put on one line, but \((+ \ a \ (- \ c \ d) \ e)\) should be broken up as:

\[
(+ \ a  
  (- \ c \ d)  
  e)
\]

- If the expression \((E_1 \ E_2 \ldots \ E_n)\) is broken across lines, then the expressions \(E_2 \ldots E_n\) should all begin on the same column.
- Don’t indent code so deeply that it runs off the end of the screen. If that starts to happen, break the definition up into several definitions. In general, 80 columns is usually a safe page width.

**Names should be chosen so a naïve reader’s first interpretation is always right**

Readers use the name of a variable or procedure to determine what that variable’s function is without having to check its definition. Names therefore have to be chosen carefully to avoid confusing the reader.
Suppose you have a variable that points at the window that’s used to display the current document. You would probably want to call that variable something like current-document-window. You should not call it current-document, since that would mislead a naïve reader into believing the variable was a document and not a window. However, if it’s clear from context in all the places where the name is used that it’s really a window, then current-document might actually be a better name, since it is more succinct. Adding -window would only be cluttering the screen with redundant information.

**Data are nouns, procedures are verbs, predicates are statements**

Some parts of a program describe data while others describe procedures. Data should be given names that are nouns or noun phrases, while procedures should be given names that are verbs or verb phrases. This is just common sense. A procedure that creates a window should usually be called make-window, not window. A variable that holds a window should be called window, new-window, w, etc., but not create-window.

Flags and predicates (Boolean variables and functions) should be named with statements. Thus the predicate that tells you if a window is closed should be window-closed? and the flag that tells you whether the current window is closed should be something like current-window-closed?. In Scheme, it is common practice to put a ? at the end of these variable names. In common lisp and sometimes in C++, people will use “-p” or “_p”, meaning “-predicate” to indicate that something is a predicate or flag.

**Exception 1: True functions should be named after their return values**

Some procedures, for example a procedure that opens a file, are called primarily for their effects. These, as I said, should generally have names that are verb phrases (i.e. that express some action). Thus the file opener would be called open_file, or open, or get_file, or what have you, but not file, since that wouldn’t make it clear whether it was opening a file or returning some previously opened file. However, many procedures are thought of by their designers as being like mathematical functions. Either they don’t have any side effects to their execution, or their side effects aren’t important. These functions are best named after their return values, which are usually nouns. Thus, a procedure that computes the cosine of its argument should be called \texttt{cos} or \texttt{cosine}, but not \texttt{compute\_cosine}, which is just silly. If it’s a procedure, it’s computing something, so adding “compute_” to its name is redundant and just clutters up the code.

**Exception 2: procedures treated as data can be nominalized**

Nominalized means “turned into a noun.” Suppose you have a procedure that calls make-window and you decide that you need to be able to change how windows are created depending on context. A common way to do this is to pass the procedure for making windows as a parameter. You might then call the parameter window-maker, a noun phrase, rather than make-window, a verb phrase, so as to emphasize the fact that it’s a data object that can vary from use to use.

**If you can’t think of a good name for it, it’s probably bad code**

Do not just call a variable “thingy” or “boolean” because you can’t think of a more descriptive name for it. Explain what it does to the TA or to a friend who’s a more experienced coder and see what they’d call it. If you can’t figure out how to explain to them what it does, then that is often an indication that it doesn’t really work at all. At very least, it’s probably too convoluted for anyone to figure out. You should try to think of a better way to write it.

Copyright © 1999 Ian Douglas Horswill.
Believe it or not, becoming a good programmer often involves extending your vocabulary. If you run out of ideas for a variable, try bringing up Word and using its thesaurus feature to look for related words and concepts that you can use.

**Comments are the documentation technique of last resort**

Suppose you’re writing some kind of a command processor. It reads a command, looks up the appropriate procedure to handle the command, and then runs it. Sometimes, you need to have a procedure whose purpose is to drop you out of the handler for the command and return you directly to command processor. (If you don’t know how to write one of these don’t worry, suffice it to say that it’s possible). A good programmer will call this procedure something like `abort-to-command-processor`. However, a bad programmer will call it something like `stop-thingy` and then write half a page of comments above its definition explaining that it aborts the processing of the current command and returns execution to the command processor. The problem is that half the readers of the program won’t be looking at the definition of the procedure when they want to understand it, they’ll be looking at one of the calls to it. Half of them will assume that there is some class called `thingy` and will start looking for `thingy.h` or `thingy.scm`, figuring that it’s better to figure out what `thingys` are first, before trying to figure out how `thingys` get stopped.

**Redundant comments are worse than no comments**

“Add one to x” is not a useful comment for the statement “`x += 1;`”. Many students add useless comments to their code that just get in the way of reading the code. You should only add comments when they express something that isn’t already evident from the code itself. Comments are just more code that the poor reader has to wade through, so you need to carefully balance their benefits against the cost of having to read them.