Coding Standards

Much of this handout originally written by Nick Parlante.

Landmarks in Coding Quality

Let's honestly review the conceptual landmarks most programmers use when thinking about how good their code is:

Publication Quality

Code suitable for publication in an article or textbook. Perfect algorithm choices, thoughtful overview and inline comments, perfect formatting, *all* the identifiers make sense, no coding hacks of even the slightest vulgarity. Something you'd be proud for anyone else to see—you'd include it with your Stanford application. By the way, no one writes code like this by accident. It has to be a deliberate goal from the beginning.

Peer-Review Quality

Some companies have "peer code review" meetings every once in a while where a programmer hands out printouts of whatever they've been working on and walk all their peers through it to give the gist of what's going on. Most programmers dread peer review because much of the code they whip out from day to day is a bit embarrassing. Companies like it because it scares the programmers into avoiding shamefully bad code. So peer review quality code tends to be mostly pretty good, but with occasional lapses that can be rationalized to an understanding audience- "I was going to decompose that, but I didn't have time." " oh yes, ha ha, on page 8 there are some utterly awful looking functions I whipped out to fix that VM leaking bug we had in May. I don't think it will port to the HP anyway. I think we're going to make the summer intern re-write it." "I'm going to go through and write documentation for that module and fix the variable names just as soon as I finish this current project I'm just right on the verge of finishing. Real soon now. I promise. Stop laughing at me!"

No-One-Is-Ever-Going-To-Look-At-This-Again Quality

Code that's being pounded out to solve a time-critical problem for the short term. No overview comments. A lot of one letter variable names. Indentation are probably ok just out of habit. Instead of decomposing out common functions, the programmer will just copy and paste chunks of code. Although this code was intended to be a two-week hack, sometimes such code ends up working its way into production code and plaguing programmers for generations to come.

Look-At-Me-I'm-So-Clever Quality

As with many other disciplines, some programmers can't resist the pleasure of noodling around in the obscure parts of a language. This is fairly understandable really—programming languages are arcane creatures, and there's something pleasing about exploiting some obscure construct. But it's not a useful impulse really. It tends to go something like this..."By eliminating all the function calls and using this other obscure platform specific feature that no normal person has ever heard of, I was able to reduce your whole program down to:

```
for(;P("\n"),R-;P("|"))for(e=C;e-;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

It's a full 24% faster, and it only took me a day." Efficiency is often cited. This impulse seems to come up in meetings and lectures too.

What's Impressive

Just for reference, here's the official chart of what's impressive and what's not, as published by the Opinionated Programmers Council headquartered in Geneva, Switzerland.

• Code which works for all cases	Impressive
• Code which was finished on time	Impressive
• Code you didn't write, but got stuck with somehow, and yet find easy to deal with and maintain	Very Impressive
• Elegant, clean code with the occasional bit of inspired cleverness	Impressive
• A complex and highly hacked-up and optimized computation packaged in a nice, painless-to-use abstraction	Impressive
• Nice-looking code which runs fast and ports with no work	Impressive
• Code which works on many, but not all cases	Not Impressive
• Nasty-looking code which is a pain to read and debug	Not Impressive
• Thoughtless code which took little time to write	Not Impressive
• Fast code which doesn't work quite right	Not Impressive

Stanford CS Standards

In a *Lord of the Flies*, uncivilized incarnation, many programmers will slip into No-One-Is-Ever-Going-To-Look-At-This-Again style code. Whatever takes the least time to type. But that habit doesn't translate well in real world where programs need to work, get maintained, etc. On the other end of the spectrum here at Stanford, all we do is write, read, and think about code and how to write it better. Even if rarely attained, we should at least hold out Publication Quality as the ideal to shoot for. Peer-Review quality might still get an "A". Mediocre looking code which computes the right answer should get a "B". And so on down the scale for code which doesn't quite work right.

The ideal for great looking code is quite high. Ask yourself: If you were to show your code around, is there any part which would seem like it could stand some improvement?

Write The High Quality Version First

For a hard programming project, it's tempting to use the following strategy: whip out a low-quality solution with no decomposition and one letter identifier names just to get something working. Do most of the debugging work on the low quality model. If there's time once everything is pretty much working, improve the decomposition, identifier names, etc. to get the program up to the "A" level. This strategy has been tried quite a bit, and it doesn't work. *It's easier to write it at the "A" level right from the start.* Well-decomposed, readable code is easier to test, will have fewer bugs, and what bugs there are will be better isolated and easier to track down. Realistically, most of your development time is not going to be consumed by typing in long identifier names or writing two 15-line functions instead of a 30-line one.

One-Pass Coding

It's more work to retrofit a feature at some time after the original coding. The first time you write something, you have all the details in your head. It's the easiest time to put in comments, extra features, etc. Every time you revisit the code later for debugging or revision, there's a period of re-learning while you remind yourself how the thing works. This becomes a real problem when your programs get seriously large.

For most of your code, you'd like to never look at it again seriously once it's initially written. Of course with the unpredictability of debugging and unplanned design revisions, you will certainly revisit some parts of your program. But if you choose a strategy of revising the whole thing later, then you will certainly need to relearn all of it, and that's going to take much more time than putting in the extra effort to fix things up on the first writing.