Pascal Style and Format Guide

A compiler does not care how your source code is formatted. And it finds your coding style or lack of style irrelevant. If a compiler ignores Style and Format then are those concepts important? Given the proper surrounding context, a Pascal compiler will generate the same machine code for the following two blocks of source code:

```
procedure initialize;begin bplotdetect(graphdriver,graphmode);case graphDriver
of ega64:if registerbgidriver(@egavgadriverproc)<0 then abort('EGA/VGA');cga:if
registerbgidriver(@cgadriverproc)<0 then abort('CGA');att400:if
registerbgidriver(@attdriverproc)<0 then abort('AT&T');hercmono:if
registerbgidriver(@hercdriverproc)<0 then
abort('Herc');end;initgraph(graphdriver,graphmode,'');abort('');
kplotInit;setlinestyle(solidln,0,normwidth);end;
```

-- Code block 0.0

```
{}PROCEDURE Initialize;
  BEGIN
    KPlotDetect(GraphDriver,GraphMode);

    CASE GraphDriver OF
      EGA64 :
        IF RegisterBGIdriver(@EGAVGADriverProc) < 0 THEN BEGIN
          Abort('EGA/VGA');
        END {IF};

      CGA :
        IF RegisterBGIdriver(@CGADriverProc)    < 0 THEN BEGIN
          Abort('CGA');
        END {IF};

      ATT400 :
        IF RegisterBGIdriver(@ATTDriverProc)    < 0 THEN BEGIN
          Abort('AT&T');
        END {IF};

      HercMono :
        IF RegisterBGIdriver(@HercDriverProc)   < 0 THEN BEGIN
          Abort('Herc');
        END {IF};
    END {CASE};

    InitGraph(GraphDriver,GraphMode,'');
    Abort('');
    KplotInit;

    SetLineStyle(SolidLn,0,NormWidth);
{}END {Initialize};
```

-- Code block 0.1

Is either of the two code blocks easier to read and understand? Why?

Which style would you prefer to use if you were required to pay for every “bug” found in your code?

Can your coding style and format influence the number of errors you leave in your code?

When I was in school, I wanted to learn the subject while maximizing my grades and minimizing the time I spent. In business I wanted to provide a needed service that maximized my profits while minimizing my costs.

In school if you slip a schedule or turn in a buggy project, you fail. In business, if you slip a schedule or ship a buggy product, you can (depending on your contract) be sued for non-compliance, incur increased maintenance and support costs, lose market share or go bankrupt.

Are bugs expensive?

You've graduated from the university and have been hired by Alpha Corporation, a (fictional) custom software development company. Alpha Corporation won the contract for developing a library management system for the Liberty University System. You are on the software development team. Your boss has given you the task of developing the resource scheduler.

When a patron requests a title for a specific date, the resource scheduler tries to find a copy of that title that is available for the requested date. If there are multiple copies available, the resource scheduler should schedule the copy that has been used the least.

Now assume that as you implement the resource scheduler you accidently insert a bug that causes the most commonly used copy of a title to be preferentially scheduled. The other copies of a title are scheduled only if the most commonly used copy is unavailable.

Your code compiles.

Case	Hours	Scenario
1	4	You tell your boss that you are finished and you submit your source code to the project archives.
2	5	You build a test program that calls your module. The test program compiles. When you run the test program, you note that copies are being scheduled. You tell your boss that you are finished and you submit your source code to the project archives.
3	8	You build a test program that calls your module. The test program compiles. You run the test program and study the inputs and outputs. You verify that if a title has multiple copies, some of which are in use for a requested date, an available copy is located and scheduled. You tell your boss that you are finished and you submit your source code to the project archives.
4	16	You build a test program that calls your module. The test program compiles. You run the test program with a set of test data to verify that the least used copy is scheduled. After studying the results, you find that the most commonly used copy is preferrentially scheduled. You re-examine your code, locate and remove that error. You retest and verify that the module now performs this action correctly. You tell your boss that you are finished and you submit your source code to the project archives.

Which is the cheapest course of action? Assume that your burdened salary (salary with all benefits, and employer paid costs) is $25/hour.

Assume that for cases 1, 2 and 3, your bug goes undetected and still exists in the final build that is delivered to Liberty University. After six months, the employees at the University note that some scheduled items are wearing faster than expected. After a year it becomes obvious that items are not being scheduled so as to spread the wear evenly across all available copies of a title. Liberty University submits a DR (Defect Report) and asks to be reimbursed for the items that had to be replaced early because of excessive wear.

What is now the cheapest course of action?

The bug described in the DR is added to the list of other defects in the project. In a meeting with the project manager the defects are reviewed. The scheduling defect is an obvious contract violation. A maintenance programmer is called in and given the task of verifying that the defect exists. Once located, the maintainence programmer is to report possible solutions before fixing the defect. Assume that it takes the maintenance programmer 4 days to build a test program; assemble a set of test data to exercise the bug; test

and verify that the problem actually exists and is repeatable. Assume that once located, the fix is reasonably obvious and takes only 2 hours to implement.

The maintenance programmer's recommendations are reviewed and approval is granted. The maintenance programmer removes the bug. The bug is documented and an additional test is added to the regression suite. The documentation department reviews the bug to see if there is any impact on the written manuals. The entire regression suite is run against the new build. The new build with updated documentation is submitted to Liberty University. The university runs their own acceptance tests on a isolated test system before calling the new build their production system. From the time the DR was submitted until the fix was placed online by the university was three months and cost a total of five man-months.

What is now the cheapest course of action?

Are there cheap bugs?

The cheapest bug is one that never sees the light of day.

The cost of fixing a bug increases as the point of discovery moves away from the programmer. Why?

In the previous example, it took the maintenance programmer four days and two hours to verify, locate and fix the defect. If that process had been instantaneous what would have been the effect on the schedule and cost? As the point of discovery moves away from the programmer, the actual cost of correction becomes a small part of the total cost to correct the defect.

Given that you will insert defects into your code how can you reduce the cost of those defects?

As you develop a block of source code, you should ask yourself the following questions:

- Does this code make sense in the global context?
- Does this code make sense in the local context?

Code clarity tends to follow the second law of thermodynamics -- it degrades with time. If the code is not clear while its context is fresh and vivid, it is unlikely to improve as your familiarity with its context fades. At some point, you or someone else will have to modify your code because the requirements will change or an error needs correcting. Clarity is essential to minimize the time and resources spent implementing the change.

When you discover a defect, you should ask yourself the following questions:

- How could I have avoided inserting this specific coding defect?
- What habits and practices can I use to eliminate this kind of defect before they occur?

Steve Maguire in *Writing Solid Code* suggests that you ask yourself the following two questions:

- How could I have *automatically* detected this bug?
- How could I have *prevented* this bug?

Steve also says that: *The easy answer to both questions would be "better testing," but that's not automatic, nor is it really preventative. Answers like "better testing" are so general that they have no muscle -- they're effectively worthless. Good answers to these questions result in specific techniques that eliminate the kind of bug you've just found.*

Section: Comments

A comment can span many lines, but the compiler treats it as if it were but a single space character. Thus a comment can appear between symbols and between words but not within a symbol or within a word. If a comment has no more significance to the compiler than a single space character, should you bother to comment your source code?

(Refer: Code Complete; Chap 19)

As the implementor, you can use comments to provide additional information to the maintenance or application programmer.

Prologs are comments used to document each routine. What problems exist with the following example?

```
{}FUNCTION AppendChar(aStr:STRING; aChr:CHAR):STRING;
{+-------------------------------------------------------------------------\
 | Name:      AppendChar                                                    \
 |                                                                           \
 | Purpose:   Append the specified character to the end of the string and   |
 |            return the result as the function value.                      |
 |                                                                          |
 | Algorithm: If the length of the passed string is less than the maximum   |
 |            string length, the passed character is appended to the end    |
 |            of the string and the result is returned as the function      |
 |            value.                                                       |
 |                                                                          |
 | Inputs:    aStr  -- the string to which the character is to be appended|
 |            aChr  -- the character to append to the string aStr.          |
 |                                                                          |
 | Outputs:   The function value is the appended string                     |
 |                                                                          |
 | Called by: CreateBorder, DoReport, EndPage, ForSummary, PrintHeader      |
 |                                                                          |
 | Author:    John P. Verbose                                               |
 |                                                                          |
 | Created:   2/1/86                                                        |
 |                                                                          |
 | Revised:   *                                                             |
 +-------------------------------------------------------------------------+}
  BEGIN
    AppendChar := aStr + aChar;
{}END {AppendChar};
```

- The vertical bars at either end are difficult to maintain.
- The *called by* section is difficult to maintain. Its inclusion makes code reuse more difficult because that section will have to be modified every time the function is used in new code.
- The complete boiler plate is overkill for such a trivial function
- The algorithm section is strained because it's hard to describe something as simple as adding a character to the end of a string at a level of detail that is simpler than the code itself.

McConnell lists five kinds of comments:

1. repeat of the code

A repetitious comment restates what the code does in different words. It gives the reader more to read without providing additional information. Why should repetitious comments be avoided?

2. explanation of the code

Explanatory comments are typically used to explain complicated, tricky or sensitive pieces of code. In such cases the comments are useful, but usually that is because the code is confusing. If the code is so complicated that an explanation is required, then it is almost always better to improve the code than to add an explanatory comment Make the code itself clearer and then add summary or intent comments.

3. marker in the code

Markers may be used to identify sections of code that have not been finished. For example:

{!!!-- add divide by zero test}

Finished code should never contain this kind of marker.

4. summary of the code

A summary comment distills a few lines of code into one or two sentences. They are valuable because a reader can scan them more quickly than the code itself. Summary comments are particularly useful when someone other than the code's original author tries to modify the code.

5. description of the code's intent

Such a comment explains the purpose of a section of code. Intent comments operate more at the problem level than the solution level. For example:

{ get item title from data base }

is an intent comment.

Summary

The general rules of thumb regarding comments are:

1. **Use styles that don't break down or discourage modification**
2. **Comment as you go along**
3. **Avoid self-indulgent comments**
4. **Write comments at the level of the code's intent**