Pascal Style and Format Guide

Format and Style

A consistent format and coding style helps the reader understand your code. Though the compiler does not care about such things, your boss will. Bad practices on your part will increase his cost and slip his schedule. Your co-workers will care when they have to read your code during reviews. The maintenance programmer will care when he has to maintain your code months or years after you've finished with it.

Some organizations impose format and style standards. Some standards are codified into programs that read, reformat and restyle your source code. Format consistency has the additional benefit of making automated source statistics more meaningful.

Q **What formatting rules should we use in this class?**

What do you think is appropriate?

What kind of formatting rules would tend to reduce errors during code construction and increase the ease of discovering errors while debugging?

Source code should be formatted to make it easy to read and understand. Formatting is a mechanical process that can be automated. A format determines how you layout your code. Formatting techniques include:

- indentation and white space. Proper use of indentation and white space visually suggests the level of code complexity;
- the use of upper, lower and mixed case letters. Pascal is case insensitive, but a consistent use of case aids in the identification of reserved words and other identifiers;
- the use of comment "tags." In Pascal, all blocks terminate with the reserved word END. Adding a comment tag such as "**END** *{IF}*;" or "**END** *{WHILE}*;" helps to identify the end of the block;
- and blank lines.

If format is the bird's eye view of the forest of code complexity, then style is the ground level signposts that help you find your way. Good style is not as mechanical as good format – it is not as easy to automate. Coding style includes the use of:

- less ambiguous language subsets;
- naming conventions; and
- appropriate comments and prologs.

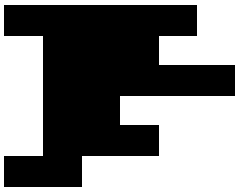
Code Format

With a good format, you can tell your readers a lot about your code – even if they never read a single line. A bad format, on the other hand, will suggest one level of complexity while the actual code tells the compiler something entirely different. The greater the difference between what your format suggests and your code instructs, the more difficult your code will be to read and maintain.

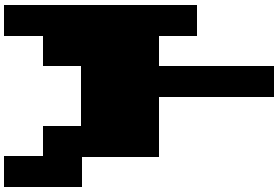
Section: Indentation and White Space

If left justification suggests a simple sequence of statements:

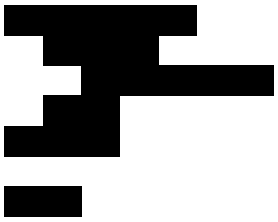
Then what does the following level of indentation imply?



Multiple levels of indentation suggest what?



And what does the following suggest?



What is wrong with the following example?

```
PROCEDURE clipscene ;
VAR i,clipindex : integer ;
BEGIN
  ppd := 3.0*horiz ;
  FOR i := 1 TO nof
  DO BEGIN
       clip(i,clipindex) ;
       IF clipindex = 3
       THEN nfac[i] := ntf
       ELSE IF clipindex = 2
            THEN nfac[i] := 0
    END
END ; { of clipscene }
```

Graphically, the indentation is:

The ClipScene procedure comes from a text book for developing high-resolution computer graphics in Pascal. I find the format difficult to read.

1. the level of indentation is visually inconsistent. Code is indented to the "G" in the reserved word **BEGIN** or the "F" in the reserved word **IF**. Worse, the "begin" and the "if" are not the first words in the line, so the physical indentation varies from 2 to 7 to 13 spaces.
2. the outdented end does not visually close the block

Using the same code (after closing the final "if"), and applying a different format, we have the following

```
PROCEDURE ClipScene;
VAR
  i,
  clipindex    : INTEGER;
BEGIN
  ppd := 3.0*horiz;
  FOR i := 1 TO nof DO BEGIN
    Clip(i,clipindex);
    IF clipindex = 3 THEN
      nfac[i] := ntf
    ELSE
    IF clipindex = 2 THEN BEGIN
      nfac[i] := 0
    END {IF};
  END {FOR};
END {ClipScene};
```

Which graphically is:

Section: Capitalization and Case

All the identifiers in the *Computer Graphics* text are lowercase. The resulting code is difficult to read. You should use upper, lower and mixed case to visually tag the different kind of identifiers. When reading your code, it should be obvious to the reader that identifiers for functions are visually distinct from the identifiers for variables or constants.

For this class, use the following conventions:

Type	Example	Convention
constants	chSpace cLightSpeed	prefix character constants with "ch" prefix all other constants with "c"
functions & procedures	GetName	Use initial upper case for Function and Procedure identifiers
private fields in class & object definitions	FsLastName	prefix the field name with "F" (Borland convention)
reserved words	BEGIN	Use all upper case for reserved words
type declarations	TEmployee	prefix the type with "T" (Borland convention)
typed constants	liCount	Same as variables (a typed constant is a variable with an initial value)
variables	lsName	Use initial lower case for variable identifiers

Section: Blank lines

Blank lines visually separate blocks of code.

For this class, use the following conventions:

Convention	Example

1 blank line after an "END" that is not followed by an "ELSE," or another "END"	`IF x THEN BEGIN`
	`  IF y THEN BEGIN`
	`  IF z THEN BEGIN`
	`  {statements if x, y & z are true}`
	`  END ELSE BEGIN`
	`  {statements if x & y are true & z is false}`
	`  END {IF};`
	`  END {IF};`
	`END {IF};`
	
	`WriteLn(lTXT,'X is: ',BoolToStr(x));`
	`WriteLn(lTXT,'Y is: ',BoolToStr(y));`
	`WriteLn(lTXT,'Z is: ',BoolToStr(z));`

2 blank lines after the semicolon in a uses list.	`USES`
	`  WinAPI, WinCRT, WinPrn, ColorDlg, CommDlg;`
	
	`TYPE`

	`TPosition =(_Hourly, _Professional, _MdlManagement,_UprManagement);`

2 blank lines between nested procedures or functions	`{}PROCEDURE Alpha;`
	`{}{}PROCEDURE Beta;`
	`BEGIN`
	`{body of procedure Beta}`
	`{}{}END {Beta};`
	
	`{}{}PROCEDURE Delta;`
	`BEGIN`
	`{body of procedure Delta}`
	`{}{}END {Delta};`
	
	`BEGIN`
	`Beta;`
	`Delta;`
	`{}END {Alpha};`

4 blank lines between program level or unit level procedures & functions	`{}PROCEDURE One;`
	`BEGIN`
	`{body of procedure One}`
	`{}END {One};`
	
	`{}PROCEDURE Two;`
	`BEGIN`
	`{body of procedure Two}`
	`{}END {Two};`

Coding Style

Programming is a mental activity with two audiences – the human reader and the machine. While good format suggests the big picture of your code to the reader, good style helps him navigate his way through the details. Good style consists of conventions that help the reader understand your code on a line-by-line basis.

Section: Language Restrictions

Computer languages are not perfect. There are constructs that while understood by the author and the compiler, may be misleading to the reader. Consider the following code fragment:

```
IF TransactionCompleted THEN
  ReportTransaction(laInvoice);
  CloseTransaction(laInvoice);
  FreeTransactionMemory(laInvoice);
```

What does the indentation suggest?

What will the compiler do if the function *TransactionCompleted* returns true? If it returns false?

What did the author intend?

A:

```
IF TransactionCompleted THEN BEGIN
  ReportTransaction(laInvoice);
END {IF};

CloseTransaction(laInvoice);
FreeTransactionMemory(laInvoice);
```

B:

```
IF TransactionCompleted THEN BEGIN
  ReportTransaction(laInvoice);
  CloseTransaction(laInvoice);
END {IF};

FreeTransactionMemory(laInvoice);
```

C:

```
IF TransactionCompleted THEN BEGIN
  ReportTransaction(laInvoice);
  CloseTransaction(laInvoice);
  FreeTransactionMemory(laInvoice);
END {IF};
```

In the above example, we can determine what the compiler will do (case A). But we do not know the author's intent. Were the procedure calls CloseTransaction and FreeTransactionMemory indented deliberately and the code written incorrectly? Or was the code written correctly and the procedure calls accidentally indented? We don't know. We cannot know. And because we don't know, maintenance becomes… ***tricky***. Intense study of the surrounding code may clarify what should occur.

Analogy If the car you are driving has a top speed of 165 mph, should you drive that fast on city streets? Just because you can does not mean you should or that it is wise to do so.

Always show your intent by using complete blocks.

Avoid	Use

`IF x<10 THEN x:=x+1;`	`IF x < 10 THEN BEGIN`
	`  x := x + 1;`
	`END {IF};`

`IF x<10 THEN x:=x+1 ELSE x:=10;`	`IF x < 10 THEN BEGIN`
	`  x := x + 1;`
	`END ELSE BEGIN`
	`  x := 10;`
	`END {IF};`

`FOR j:=1 TO 10 DO DoSomething(j);`	`FOR j := 1 TO 10 DO BEGIN`
	`  DoSomething(j);`
	`END {FOR};`

`WHILE x<10 DO x:=x+1;`	`WHILE x < 10 DO BEGIN`
	`  x := x+1;`
	`END {WHILE};`

`WITH aPerson DO Write(lastName);`	`WITH aPerson DO BEGIN`
	`  Write(lastName);`
	`END {WITH};`

The only advantage the code in the left column has over the code in the right is that it is textually shorter. It is not faster. It is not easier to understand. It is not easier to modify. Use full blocks and stay out of trouble.

Section: Naming Conventions

Identifiers denote constants, types, variables, procedures, functions, units, programs and fields (in records, objects and classes).

The accepted Borland naming conventions are the following:

- user defined type definitions are prefixed by a capital T
- private (and protected [Delphi]) fields are prefixed by a capital F.

For this class, also use the following variant of Hungarian notation:

Convention	Scope of variable	Explanation
aA	Procedure or Function formal arguments	Use a "a" to indicate that the variable is a formal argument
gA	Visible Unit global variable	Use a "g" to indicate that a variable is global to any program (or other unit) that contains the unit in its USES list.
lA	Local variables	Use a "l" (lower case L) to indicate that the variable is local to the procedure or function
uA	Global variable that follows the IMPLEMENTATION reserved word	Use a "u" to indicate that the variable is global to the body of a unit and not visible outside the unit.

Convention	Variable Type	Explanation and example

xaA	Array variables	**TYPE**
		  TStudentName : **STRING**[50];
		VAR
		  uaStudents : **ARRAY**[1..cMaxStudents]**OF** TStudentName;

xbA	Boolean type variables	BOOLEAN, ByteBOOL, LongBOOL, WordBOOL **VAR** lbHomeworkCompleted : BOOLEAN;
xcA	Character variables	**VAR** lcNextChar : CHAR;

xeA	Enumerated variables	**TYPE**
		  TPrimaryColors=(_red,_green,_blue);
		VAR
		  geColors : TprimaryColors;

xfA	Floating point variables	Assignment compatible floating point types: COMP, CURRENCY (Delphi), DOUBLE, EXTENDED, REAL, SINGLE

		VAR lfAtomicMass : DOUBLE;
xiA	Integer type variables	Assignment compatible integer types: BYTE, CARDINAL, INTEGER, LongINT, ShortINT, SmallINT (Delphi), WORD **VAR** giNoOfStudents : INTEGER;
xoA	Class or Object variables	**VAR** loForm : TForm;
xpA	pointer variables	**VAR** lpStudentName : ^TStudentName;

xrA	Record variables	**TYPE**
		  TPoint = **RECORD**
		  x : INTEGER;
		  y : INTEGER;
		  END *{RECORD}*;
		
		VAR
		  grPoint : TPoint;

xsA	String type variables	AnsiSTRING (Delphi), ShortSTRING (Delphi), STRING **PROCEDURE** PrntName(aLastName : **STRING**);

Section: Comments

Use the following conventions:

- For normal comments use the brace-style comments, the "{this is a comment}." If you comment out blocks of code, use the parenthesis-asterisk comment, the "(* this is also a comment *)."
- Use "End-tag-comments"
- Use "bubbles" (empty comments) for proper indentation of nested functions or procedures. For example:

```
{}PROCEDURE Alpha;
{}{}PROCEDURE Beta;
    BEGIN
      {body of procedure Beta}
{}{}END {Beta};

(*
{}{}PROCEDURE Delta;
    BEGIN
      {body of procedure Delta}
{}{}END {Delta};
*)

  BEGIN
    Beta;
(*  Delta;*)
{}END {Alpha};
```

In the sample code shown above:

1. *{Alpha}*, *{Beta}* and *{Delta}* are end tag comments – they identify the block being completed.

2. The Delta procedure has been commented out using *(* and *)*
3. Bubbles ("*{}*") are used to ensure proper indentation of nested procedures. Both "Beta" and "Delta" are procedures nested inside of "Alpha." A sequence of bubbles (with one bubble for each level of nesting) is placed to the left of the procedure heading. Use the same number of bubbles before the "end" that terminates the procedure body.

- Source files should contain a prolog similar to the following:

```
UNIT RsrcSchd;
{+H
-------------------------------------------------------------------
  File          - RSRCSCHD.PAS
  Purpose       - Resource Scheduler for Liberty University Contract
                  87A5. Preferentially schedule the least used copy
                  of a requested resource.
  Copyright     © 1987 Alpha Corp. (except where otherwise noted)
                  All rights reserved.
  Author        - Will E. Makit
  Revised       - 1987.0215 (WEM) Wrote original code.
                - 1987.0225 (WEM) Revised scheduling algorithm.
                  See notes in GetLeastUsedCopy.
                - 1987.1204 (KSB) Rewrote TestForConflicts. See notes
                  in procedure.
-------------------------------------------------------------------}
INTERFACE
USES
  Glbl_LUC,  DB_Title, DB_Copy, Calendar;


TYPE
  {global type declarations}
CONST
  {global constant declarations}
VAR
  {global variable declarations}

{}PROCEDURE   GetLeastUsedCopy(aCopyList:TCopyList; VAR aLUcopy:Tcopy);
{}PROCEDURE   RequestCopyOf(aTitle:Ttitle; VAR aCopyList:TCopyList);
{}FUNCTION    TestForConflicts(aLUcopy:Tcopy):TTestStatus;
               {=============================================}
IMPLEMENTATION
  {body of unit; RequestCopyOf, GetLeastUsedCopy and TestForConflicts
   are implemented here}
END {RsrcSchd}.
```

CAUTIONS

When you write source code, you are writing a public document. Your goal is to communicate clearly and effectively to two very different audiences; the compiler and the code reader. Odds are that someone, some day will read the code you write. That person may be a co-worker, your boss, a review board, a lawyer or your mother. Therefore consider the following when writing code:

- Do not use offensive words as identifiers.

- Do not include offensive remarks as comments.
- Don't be an "e. e. cummings" while writing code – save that kind of creativity for your friends and family. Good code has its own beauty, but it is not poetry.

Summary

A set of conventions is one of the intellectual tools used to manage complexity - McConnell.

Much of the detail associated with programming is arbitrary. How do you format a comment? How many spaces should you indent the body of a block? How should declarations be ordered – or should they be ordered? Questions like these have more than one right answer. The convention used is less important than that it be used consistently. Conventions save you the trouble of answering the same arbitrary questions repeatedly. On projects with many programmers, conventions prevent confusion when different programmers make individual choices for the arbitrary decisions.

A convention conveys important information concisely. In naming conventions, a single letter can differentiate among formal arguments, local, unit implementation and global variables. Capitalization can separate variables and constants from functions and procedures. Indentation conventions can convey the logical structure of a procedure, function or main program block.

Conventions can protect against known hazards. You can establish conventions to eliminate the use of dangerous practices or to restrict their use to cases in which they are truly needed. You should use the convention that prohibits the following dangerous practices: multiple declarations on one line; and multiple statements on one line. You should compensate for dangerous (or poorly remembered) operator precedence rules by requiring parenthesis around complicated expressions.

With good conventions, more can be taken for granted as fewer details will have to be absorbed for program comprehension. Conventions add predictability. Having convention ways of handling common tasks adds meaningful structure to your program and makes it easier for another programmer familiar with your conventions to understand.

It is possible to go overboard with conventions. A programmer could have so many standards, guidelines and rules of thumb that remembering them is a full-time job. However, programmers on small projects usually suffer from the opposite problem – insufficient standards, not realizing the benefits of intelligently constructed conventions.