Pascaline Language Coding Standard

Contents

1 Purpose and scope

This document contains a coding standard for Pascaline based programs.

Pascal/Pascaline already enforces good style via its syntax. There also are good practices concerning documentation, indentation, etc.

2 Rules

2.1 Module format

Each module is contained in its own file.

Each module shall have the following format:

1. The module comment header.
2. The joins statement, if exists.
3. The uses statement if exists.
4. The preferred order of declarations is the Pascal order, constants, types, fixed, variables, procedures and functions, then the program or module block followed by the finalization block if exists.

2.2 Module heading

Each module shall have a heading without exception. The heading shall appear:

```
{****************************************************************

Single line description

Extended description (multiline as required).

******************************************************************/
```

2.3 Procedure and Function heading

Each procedure and function shall have a heading unless:

1. It is in a group of related functions that vary from each other trivially, and a header exists for all of them.
2. It is a function only used within another function (it is essentially nested within that function). In this case it shall appear after the function heading comment for the including function and be documented by that function heading.

The function heading shall appear as:

```
{*************************************************************

Single line description

Extended description (multiple lines if required).

**************************************************************}
```

The parameters and return value use the “prefix” format, meaning they proceed the declaration. This means the documentation program can apply the last comment to the declaration. This rule applies to all comments except for the module header.

```
procedure myfunc(
    { number of times to perform } count: integer;
    { data value }                 data:  string):
    { returns last value }         byte;
```

2.4 Prefix comments

All comments for:

- Constants.
- Variables.
- Types.
- Fixed.

Shall be prefixed commented.

2.5 Indentation

Indentation shall be done using 4 (four) spaces.

No tabs are ever to be used in source code. There are no exceptions.

2.6 Code blocks

All code blocks are indented:

```
if x begin

    Perform(y);

end;
```

All code blocks will start with a blank line and end with a blank line.

2.7 Follow on statements or code

All statements or code that follows other code shall be indented. If the follow on is a statement, then it shall be indented 4 spaces:

```
if x then
    perform(y);
```

If the follow on code is a parameter or expression part, it shall be indented FOLLOWING THE PARENTHSIS THAT STARTED IT:

```
if a or (b and
         c) then
    perform(y);
```

or

```
perform(x,
        y,
        z);
```

2.8 80 character limit

A limit of 80 characters shall apply to all source files. No line shall exceed 80 characters in width. Statements shall be broken into sections and indented as required. Strings are broken into concatenated sections such as:

2.9 Spacer lines

A blank line shall appear:

1. Before and after each function heading (the name, return specification and the parameters).
2. Before and after each block bracket (“begin“, and “end”) unless they are on the same line.
3. Before the result statement in a function.
4. After the function declaration section and before the code in the block appears.
5. After each function block ends.

This program illustrates the formatting:

```
{*******************************************************************

Copy program

Copies one text file to another

Usage:

copy filea fileb

Copies filea to fileb

*******************************************************************}

program copy(filea, fileb);

var

    { input file }  filea: text;
    { output file } fileb: text;

{*******************************************************************

Copy

Copies text lines from one file to another.

*******************************************************************}

procedure copyfile(
    { input file } var a: text;
    { output file } var b: text);

var

     { character buffer } c: char;

begin

    while not eof(a) do begin

        while not eoln(a) do begin

            read(a, c);
            write(b, c)

        end else begin

            readln(a);
            writeln(b)
        end

    end

end;

begin { copy }

    reset(filea);
    rewrite(fileb);
    copyfile(filea, fileb)

end.
```

2.10 No gotos (for most code)

The goto statement should be used. If you think you have to have a goto in a given function, this usually indicates your function is too large and needs to be broken into smaller functions.

The major use of gotos is interprocedure/interfunction gotos. In Pascal this was necessary to accomplish bailouts from deep nested procedures or functions. The preferred method in Pascaline is the use of exceptions or overrides. In general intraprocedure gotos can be used in the same module/file, but use exceptions or overrides between modules/files.

2.11 Size of Procedures and Functions

Procedures and functions should be no more than one or two pages long, where a page is defined as about 60 lines long. If the function is longer than that, there should be a good fundamental reason for it, such as needing to express a unusually large case statement. A CPU emulator routine would be an example of such a routine.

2.12 No reliance on initializations

The code shall not rely on the fact that the variables are initialized to zero. This means both that the code is restartable (can be restarted from the main() function) without clearing out the .bss section, and makes it explicit what values are being relied upon in the code.

2.13 Use break character to divide segmented names

Use the "_" character to separate parts of a symbol:

my_special_function

NOT:

MySpecialFunction

2.14 Use break character for large numbers

Very large numbers can be divided into sections:

const very_large_number = 1_000_000;

The standard method is to use 3 digit sections for decimal numbers, vs. 4 digit sections for hexadecimal numbers:

const big_address = $100_0000;

2.15 No reliance on undefined or implementation specific behavior

Features listed in the standard as undefined or implementation defined are not be used. Exceptions include features clearly meant to be implementation defined, such as the maxium value of integer (maxint), etc.

2.16 English comments

Comments shall be in English.

2.17 ISO 8859-1 character set

The ISO 8859-1 character set shall be used in code and comments without exception.

The name of the structure within the structure shall be prepended with an underscore ("_"). This name shall not appear anywhere outside of the structure definition. The structure name shall be coined from the typedef name, if there is one.

2.18 Comment copiously

Functions without comment headers or function comment headers without adequate descriptions are not allowed. Modules without adequate comment headers describing their contents, purpose and all of the exposed globals are not allowed.

2.19 Use asserts constantly

Asserts are free, in terms of the fact that they can be automatically removed on compiler option. They should be used anytime an invalid condition could occur.

2.20 Do not use asserts for runtime faults

Asserts should never be used for possible runtime faults. That would mean that code with asserts turned off would crash at runtime. Use exceptions instead.

2.21 Don't use exceptions for regular code

Exceptions are expensive in terms of runtime. The should not be used in the normal path of code, but only to handle exceptional conditions (hence the name). Usually this happens when the program terminates. There are exceptions. For example, a command line processor would use exceptions to bail out of error conditions to the main command processor without exiting the program.

2.22 Prefer use of dynamic allocation instead of static

Large structures or structures with size that varies according to use should be allocated dynamically. Code should not fail simply because a fixed table size was exceeded. The maximum limit of allocation should, in virtually all cases, be the amount of memory available to the program, not an arbitrary limit choosen by the programmer.

2.23 Code reviews

All code should be peer reviewed by at least one other programmer. Typically, the branch the new code lives on is examined by the reviewer(s), using an appropriate visual difference tool.

2.24 Source control

All source is placed under source control. We will use GIT because of its branching capabilities. Every change is to be a branch, and all changes, even in progress, are to be visible to other programmers. There is no “hidden” or private code. No work in progress is to be left out of source control for either more than one day or more than one page.

2.25 Commit early and commit often

Prefer short feature commits vs. very large commits. Prefer committing to mainline vs. branching.

2.26 Don’t commit code that does not compile

QED. If you must commit such code, make an explicit commit message that states the code is not compilable.

2.27 Hanging vs. pinned blocks

We don’t have a particular style preference. The two major styles in use are hanging or pinned block starts:

```
if true then begin

    dothis

end else begin

    dothat

}
```

for hanging brackets vs:

```
if true then
begin

    dothis

end else begin

    dothat

end
```

However the styles should be used consistently in the same program, so that the reader learns to expect a particular appearance of the code.

2.28 Run-on vs blocked code

There is no preference between these styles:

```
if true then dothis;
```

For run-on vs:

```
if true then begin

    dothis

end;
```

Obviously run-on lines only work for a single statement.

2.29 Use natural definition order

Optimally, functions in the same module should appear so that the functions that use another function appear after it. Many programmers find this style unnatural since it forces the placement of smaller, utility routines first in the file, followed by increasingly larger routines, such as main(). However, the opposite style forces predeclaration of everything. Natural definition is the style in use here. This ordering becomes second nature rapidly. It also cuts down on forward declared procedures and functions.