Index

- `style.adb` file: Program Structure
- `style.adb` file: Lexical Elements
- Subprogram bodies: Subprograms
- Subprograms: Subprograms
- Underscores: Lexical Elements
- `use` clauses: Packages

Table of Contents

2 Lexical Elements

2.1 Character Set and Separators

- The character set used should be plain 7-bit ASCII. The only separators allowed are space and the end-of-line sequence. No other control character or format effector (such as `HT`, `VT`, `FF`) should be used. The normal end-of-line sequence is used, which may be `LF`, `CR/LF` or `CR`, depending on the host system. An optional `SUB` (`16#1A#`) may be present as the last character in the file on hosts using that character as file terminator.
- Files that are checked in or distributed should be in host format.
- A line should never be longer than 79 characters, not counting the line separator.
- Lines must not have trailing blanks.
- Indentation is 3 characters per level for `if` statements, loops, and `case` statements. For exact information on required spacing between lexical elements, see file `style.adb`.

2.2 Identifiers

- Identifiers will start with an upper case letter, and each letter following an underscore will be upper case. Short acronyms may be all upper case. All other letters are lower case. An exception is for identifiers matching a foreign language. In particular, we use all lower case where appropriate for C.
- Use underscores to separate words in an identifier.
- Try to limit your use of abbreviations in identifiers. It is ok to make a few abbreviations, explain what they mean, and then use them frequently, but don't use lots of obscure abbreviations. An example is the `ALI` word which stands for Ada Library Information and is by convention always written in upper-case when used in entity names.

```
procedure Find_ALI_Files;
```

- Don't use the variable name `I`, use `J` instead; `I` is too easily confused with `1` in some fonts. Similarly don't use the variable `O`, which is too easily mistaken for the number `0`.

2.3 Numeric Literals

- Numeric literals should include underscores where helpful for readability.

```
1_000_000
16#8000_000#
3.14159_26535_89793_23846
```

2.4 Reserved Words

- Reserved words use all lower case.

```
return else
```

- The words `Access`, `Delta` and `Digits` are capitalized when used as `attribute_designator`.

2.5 Comments

- A comment starts with `--` followed by two spaces. The only exception to this rule (i.e. one space is tolerated) is when the comment ends with a single space followed by `--`. It is also acceptable to have only one space between `--` and the start of the comment when the comment is at the end of a line, after some Ada code.
- Every sentence in a comment should start with an upper-case letter (including the first letter of the comment).
- When declarations are commented with “hanging” comments, i.e. comments after the declaration, there is no blank line before the comment, and if it is absolutely necessary to have blank lines within the comments, e.g. to make paragraph separations within a single comment, these blank lines *do* have a `--` (unlike the normal rule, which is to use entirely blank lines for separating comment paragraphs). The comment starts at same level of indentation as code it is commenting.

```
z : Integer;
--  Integer value for storing value of z
--
```

```
--  The previous line was a blank line.
```

- Comments that are dubious or incomplete, or that comment on possibly wrong or incomplete code, should be preceded or followed by `???`.
- Comments in a subprogram body must generally be surrounded by blank lines. An exception is a comment that follows a line containing a single keyword (`begin`, `else`, `loop`):

```
begin
   --  Comment for the next statement

   A := 5;

   --  Comment for the B statement

   B := 6;
end;
```

- In sequences of statements, comments at the end of the lines should be aligned.

```
My_Identifier := 5;      --  First comment
Other_Id := 6;           --  Second comment
```

- Short comments that fit on a single line are *not* ended with a period. Comments taking more than a line are punctuated in the normal manner.
- Comments should focus on *why* instead of *what*. Descriptions of what subprograms do go with the specification.
- Comments describing a subprogram spec should specifically mention the formal argument names. General rule: write a comment that does not depend on the names of things. The names are supplementary, not sufficient, as comments.
- *Do not* put two spaces after periods in comments.

3 Declarations and Types

- In entity declarations, colons must be surrounded by spaces. Colons should be aligned.

```
Entity1   : Integer;
My_Entity : Integer;
```

- Declarations should be grouped in a logical order. Related groups of declarations may be preceded by a header comment.
- All local subprograms in a subprogram or package body should be declared before the first local subprogram body.
- Do not declare local entities that hide global entities.
- Do not declare multiple variables in one declaration that spans lines. Start a new declaration on each line, instead.
- The `defining_identifiers` of global declarations serve as comments of a sort. So don't choose terse names, but look for names that give useful information instead.
- Local names can be shorter, because they are used only within one context, where comments explain their purpose.

4 Expressions and Names

- Every operator must be surrounded by spaces. An exception is that this rule does not apply to the exponentiation operator, for which there are no specific layout rules. The reason for this exception is that sometimes it makes clearer reading to leave out the spaces around exponentiation.

```
E := A * B**2 + 3 * (C - D);
```

- Use parentheses where they clarify the intended association of operands with operators:

```
(A / B) * C
```

5 Statements

5.1 Simple and Compound Statements

- Use only one statement or label per line.
- A longer `sequence_of_statements` may be divided in logical groups or separated from surrounding code using a blank line.

5.2 If Statements

- When the `if`, `elsif` or `else` keywords fit on the same line with the condition and the `then` keyword, then the statement is formatted as follows:

```
if condition then
   ...
elsif condition then
   ...
else
   ...
end if;
```

When the above layout is not possible, `then` should be aligned with `if`, and conditions should preferably be split before an `and` or `or` keyword a follows:

```
if long_condition_that_has_to_be_split
  and then continued_on_the_next_line
then
   ...
end if;
```

The `elsif`, `else` and `end if` always line up with the `if` keyword. The preferred location for splitting the line is before `and` or `or`. The continuation of a condition is indented with two spaces or as many as needed to make nesting clear. As an exception, if conditions are closely related either of the following is allowed:

```
if x = lakdsjfhlkashfdlkflkdsalkhfsalkdhflkjdsahf
     or else
   x = asldkjhalkdsjfhhfd
     or else
   x = asdfadsfadsf
```

```
then
  ...
end if;

if x = lakdsjfhlkashfdlkflkdsalkhfsalkdhflkjdsahf or else
   x = asldkjhalkdsjfhhfd                          or else
   x = asdfadsfadsf
then
  ...
end if;
```

- Conditions should use short-circuit forms (`and then`, `or else`), except when the operands are boolean variables or boolean constants.
- Complex conditions in `if` statements are indented two characters:

```
if this_complex_condition
  and then that_other_one
  and then one_last_one
then
   ...
end if;
```

  There are some cases where complex conditionals can be laid out in manners that do not follow these rules to preserve better parallelism between branches, e.g.

```
if xyz.abc (gef) = 'c'
     or else
   xyz.abc (gef) = 'x'
then
   ...
end if;
```

- Every `if` block is preceded and followed by a blank line, except where it begins or ends a `sequence_of_statements`.

```
A := 5;

if A = 5 then
   null;
end if;

A := 6;
```

5.3 Case Statements

- Layout is as below. For long `case` statements, the extra indentation can be saved by aligning the `when` clauses with the opening `case`.

```
case expression is
   when condition =>
      ...
   when condition =>
      ...
end case;
```

5.4 Loop Statements

When possible, have `for` or `while` on one line with the condition and the `loop` keyword.

```
for J in S'Range loop
   ...
end loop;
```

If the condition is too long, split the condition (see "If statements" above) and align `loop` with the `for` or `while` keyword.

```
while long_condition_that_has_to_be_split
  and then continued_on_the_next_line
loop
   ...
end loop;
```

If the `loop_statement` has an identifier, it is laid out as follows:

```
Outer : while not condition loop
   ...
end Outer;
```

5.5 Block Statements

- The `declare` (optional), `begin` and `end` words are aligned, except when the `block_statement` is named. There is a blank line before the `begin` keyword:

```
Some_Block : declare
   ...

begin
   ...
end Some_Block;
```

6 Subprograms

6.1 Subprogram Declarations

- Do not write the `in` for parameters.

```
function Length (S : String) return Integer;
```

- When the declaration line for a procedure or a function is too long to fit the entire declaration (including the keyword procedure or function) on a single line, then fold it, putting a single parameter on a line, aligning the colons, as in:

```
procedure Set_Heading
   (Source : String;
    Count  : Natural;
    Pad    : Character := Space;
    Fill   : Boolean   := True);
```

In the case of a function, if the entire spec does not fit on one line, then the return may appear after the last parameter, as in:

```
function Head
  (Source : String;
   Count  : Natural;
   Pad    : Character := Space) return String;
```

Or it may appear on its own as a separate line. This form is preferred when putting the return on the same line as the last parameter would result in an overlong line. The return type may optionally be aligned with the types of the parameters (usually we do this aligning if it results only in a small number of extra spaces, and otherwise we don't attempt to align). So two alternative forms for the above spec are:

```
function Head
  (Source : String;
   Count  : Natural;
   Pad    : Character := Space)
   return   String;

function Head
  (Source : String;
   Count  : Natural;
   Pad    : Character := Space)
   return String;
```

6.2 Subprogram Bodies

- Function and procedure bodies should usually be sorted alphabetically. Do not attempt to sort them in some logical order by functionality. For a sequence of subprogram specs, a general alphabetical sorting is also usually appropriate, but occasionally it makes sense to group by major function, with appropriate headers.
- All subprograms have a header giving the function name, with the following format:

```
-----------------
-- My_Function --
-----------------

procedure My_Function is
begin
   ...
end My_Function;
```

Note that the name in the header is preceded by a single space, not two spaces as for other comments. These headers are used on nested subprograms as well as outer level subprograms. They may also be used as headers for sections of comments, or collections of declarations that are related.

- Every subprogram body must have a preceding `subprogram_declaration`.
- A sequence of declarations may optionally be separated from the following begin by a blank line. Just as we optionally allow blank lines in general between declarations, this blank line should be present only if it improves readability. Generally we avoid this blank line if the declarative part is small (one or two lines) and the body has no blank lines, and we include it if the declarative part is long or if the body has blank lines.
- If the declarations in a subprogram contain at least one nested subprogram body, then just

before the `begin` of the enclosing subprogram, there is a comment line and a blank line:

```
--  Start of processing for Enclosing_Subprogram

begin
  ...
end Enclosing_Subprogram;
```

- When nested subprograms are present, variables that are referenced by any nested subprogram should precede the nested subprogram specs. For variables that are not referenced by nested procedures, the declarations can either also be before any of the nested subprogram specs (this is the old style, more generally used). Or then can come just before the begin, with a header. The following example shows the two possible styles:

```
procedure Style1 is
   Var_Referenced_In_Nested      : Integer;
   Var_Referenced_Only_In_Style1 : Integer;

   proc Nested;
   --  Comments ...


   ------------
   -- Nested --
   ------------

   procedure Nested is
   begin
      ...
   end Nested;

--  Start of processing for Style1

begin
   ...
end Style1;

procedure Style2 is
   Var_Referenced_In_Nested : Integer;

   proc Nested;
   --  Comments ...

   ------------
   -- Nested --
   ------------

   procedure Nested is
   begin
      ...
   end Nested;

   --  Local variables

   Var_Referenced_Only_In_Style2 : Integer;

--  Start of processing for Style2

begin
   ...
end Style2;
```

For new code, we generally prefer Style2, but we do not insist on modifying all legacy occurrences of Style1, which is still much more common in the sources.

7 Packages and Visibility Rules

- All program units and subprograms have their name at the end:

```
package P is
   ...
end P;
```

- We will use the style of `use`-ing `with`-ed packages, with the context clauses looking like:

```
with A; use A;
with B; use B;
```

- Names declared in the visible part of packages should be unique, to prevent name clashes when the packages are `use`d.

```
package Entity is
   type Entity_Kind is ...;
   ...
end Entity;
```

- After the file header comment, the context clause and unit specification should be the first thing in a `program_unit`.
- Preelaborate, Pure and Elaborate_Body pragmas should be added right after the package name, indented an extra level and using the parameterless form:

```
package Preelaborate_Package is
   pragma Preelaborate;
   ...
end Preelaborate_Package;
```

8 Program Structure and Compilation Issues

- Every GNAT source file must be compiled with the `-gnatg` switch to check the coding style. (Note that you should look at `style.adb` to see the lexical rules enforced by `-gnatg`).
- Each source file should contain only one compilation unit.
- Filenames should be 8 or fewer characters, followed by the `.adb` extension for a body or `.ads` for a spec.
- Unit names should be distinct when “krunch”ed to 8 characters (see `krunch.ads`) and the filenames should match the unit name, except that they are all lower case.