GNAT Coding Style

A guide for GNAT developers

Ada Core Technologies, Inc.

1 General

Most of GNAT is written in Ada using a consistent style to ensure readability of the code. This document has been written to help maintain this consistent style, while having a large group of developers work on the compiler.

For the coding style in the C parts of the compiler and run time, see the GNU Coding Guidelines.

This document is structured after the Ada Reference manual. Those familiar with that document should be able to quickly lookup style rules for particular constructs.

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, 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 '`I`', use '`J`' instead, '`I`' is too easily mixed up with '`1`' in some fonts. Similarly don't use the variable '`O`', which is too easily mixed up with '`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

- Comment start with '`-- `' (i.e. '`--`' followed by two spaces). The only exception to this rule (i.e. one space is tolerated) is when the comment ends with '` --`'. It also accepted 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 these blank lines *do* have a '`--`' (unlike the normal rule, which is to use entirely blank lines for separating comment paragraphs). The comment start at same level of indentation as code they are commenting.

```
z : Integer;
--  Integer value for storing value of z
--
--  The previous line was a blank line.
```

- Comments that are dubious or incomplete or 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, except after a '`begin`':

```
begin
   --  Comment for the next statement

   A := 5;

   --  Comment for the B statement

   B := 6;
```

- 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.
- Don't declare local entities that hide global entities.
- Don't declare multiple variables in one declaration that spans lines. Start a new declaration on each line, instead.
- The `defining_identifier`s 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, except for the exponentiation operator.

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

- When folding a long line, fold before an operator, not after.
- Use parentheses where they make the intended order of evaluation clearer:

```
(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 exception, if conditions are closely related either of the following is allowed:

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

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

- Conditions should use short-circuit forms ('`and then`', '`or else`').
- Complex conditions in if statements are indented two characters:

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

- 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`' statements 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, especially in functions:

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

- When the declaration line for a procedure or a function is too long, fold it. In this case, align the colons, and, for functions, the result type.

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

- The parameter list for a subprogram is preceded by a space:

```
procedure Func (A : Integer);
```

6.2 Subprogram Bodies

- The functions and procedures should always be sorted alphabetically in a compilation unit.
- All subprograms have a header giving the function name, with the following format:

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

procedure My_Function is
begin
```

  Note that the name in the header is preceded by a single space, not two spaces as for other comments.
- Every subprogram body must have a preceding `subprogram_declaration`.
- If there any declarations in a subprogram, the '`begin`' keyword is preceded by a blank line.
- If the declarations in a subprogram contain at least one nested subprogram body, then just before the of the enclosing subprogram '`begin`', there is a line:

```
   --  Start of processing for Enclosing_Subprogram

begin
```

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`.

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 characters or less followed by the '`.adb`' extension for a body or '`.ads`' for a spec.
- Unit names should be distinct when krunched to 8 characters (see '`krunch.ads`') and the filenames should match the unit name, except that they are all lower case.