GNATcheck Reference Manual

Release 2019

May 18, 2019

This page is intentionally left blank.

CONTENTS

GNAT, The GNU Ada Development Environment

The GNAT Ada Compiler
Version 2019

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being 'GNU Free Documentation License', with the Front-Cover Texts being 'GNATcheck Reference Manual', and with no Back-Cover Texts. A copy of the license is included in the section entitled 'GNU Free Documentation License'.

This page is intentionally left blank.

CHAPTER

ONE

ABOUT THIS MANUAL

The *gnatcheck* tool in GNAT can be used to enforce coding conventions by analyzing Ada source programs with respect to a set of *rules* supplied at tool invocation. This manual describes the complete set of predefined rules that *gnatcheck* can take as input.

What This Manual Contains

This manual contains a description of *gnatcheck*, an ASIS-based utility that checks properties of Ada source files according to a given set of semantic rules

- *Introduction*, gives the general overview of the *gnatcheck* tool

  Format of the Report File, describes the structure of the report file generated by *gnatcheck*

  General gnatcheck Switches, describes switches that control the general behavior of *gnatcheck*

  gnatcheck Rule Options, describes options used to control a set of rules to be checked by *gnatcheck*

  Adding the Results of Compiler Checks to gnatcheck Output, explains how the results of the check performed by the GNAT compiler can be added to the report generated by *gnatcheck*

  Rule exemption, explains how to turn off a rule check for a specified fragment of a source file

  Predefined Rules, contains a description of each predefined *gnatcheck* rule, organized into categories.

  Example of gnatcheck Usage, contains a full example of *gnatcheck* usage

  List of Rules, gives an alphabetized list of all predefined rules, for ease of reference.

The name of each rule (the ‘rule identifier’) denotes the condition that is detected and flagged by *gnatcheck*. The rule identifier is used as a parameter of the `+R` or `-R` switch to *gnatcheck*.

What You Should Know Before Reading This Manual

You should be familiar with the Ada language and with the usage of GNAT in general; please refer to the GNAT User’s Guide.

This page is intentionally left blank.

CHAPTER TWO

INTRODUCTION

The *gnatcheck* tool is an ASIS-based utility that checks properties of Ada source files according to a given set of semantic rules.

In order to check compliance with a given rule, *gnatcheck* has to semantically analyze the Ada sources. Therefore, checks can only be performed on legal Ada units. Moreover, when a unit depends semantically upon units located outside the current directory, the source search path has to be provided when calling *gnatcheck*, either through a specified project file or through *gnatcheck* switches as described below.

If the set of sources to be processed by `gnatcheck` contains sources with preprocessing directives then the needed options should be provided to run preprocessor as a part of the *gnatcheck* call, and detected rule violations will correspond to preprocessed sources.

A number of rules are predefined in *gnatcheck* and are described later in this chapter. You can also add new rules, by modifying the *gnatcheck* code and rebuilding the tool. In order to add a simple rule making some local checks, a small amount of straightforward ASIS-based programming is usually needed.

Invoking *gnatcheck* on the command line has the form:

```
$ gnatcheck [switches]  {filename}
      [-files={arg_list_filename}]
      [-cargs gcc_switches] -rules rule_options
```

where

- *switches* specify the general tool options
- Each *filename* is the name (including the extension) of a source file to process. 'Wildcards' are allowed, and the file name may contain path information.
- Each *arg_list_filename* is the name (including the extension) of a text file containing the names of the source files to process, separated by spaces or line breaks.
- *gcc_switches* is a list of switches for *gcc*. They will be passed on to all compiler invocations made by *gnatcheck* to generate the ASIS trees. Here you can provide `-I` switches to form the source search path, and use the `-gnatec` switch to set the configuration file, etc.
- *rule_options* is a list of options for controlling a set of rules to be checked by *gnatcheck* (*gnatcheck Rule Options*).

Either a `filename` or an `arg_list_filename` must be supplied.

This page is intentionally left blank.

CHAPTER THREE

FORMAT OF THE REPORT FILE

The *gnatcheck* tool outputs on `stderr` all messages concerning rule violations except if running in quiet mode. By default it also creates a text file that contains the complete report of the last gnatcheck run, this file is named `gnatcheck.out`. A user can specify generation of the XML version of the report file (its default name is `gnatcheck.xml`) If *gnatcheck* is called with a project file, the report file is located in the object directory defined by the project file (or in the directory where the argument project file is located if no object directory is defined), if `--subdirs` option is specified, the file is placed in the subrirectory of this directory specified by this option. Otherwise it is located in the current directory; the `-o` or `-ox` option can be used to change the name and/or location of the text or XML report file. This text report contains:

- general details of the *gnatcheck* run: date and time of the run, the version of the tool that has generated this report, full parameters of the *gnatcheck* invocation, reference to the list of checked sources and applied rules (coding standard);
- summary of the run (number of checked sources and detected violations);
- list of exempted coding standard violations;
- list of non-exempted coding standard violations;
- list of problems in the definition of exemption sections;
- list of language violations (compile-time errors) detected in processed sources;

The references to the list of checked sources and applied rules are references to the text files that contain the corresponding information. These files could be either files supplied as *gnatcheck* parameters or files created by *gnatcheck*; in the latter case these files are located in the same directory as the report file.

The content of the XML report is similar to the text report except that it explores the set of files processed by gnatcheck and the coding standard used for checking these files.

This page is intentionally left blank.

CHAPTER

FOUR

GENERAL *GNATCHECK* SWITCHES

The following switches control the general *gnatcheck* behavior

`--version` Display Copyright and version, then exit disregarding all other options.

`--help` Display usage, then exit disregarding all other options.

`-P file` Indicates the name of the project file that describes the set of sources to be processed. The exact set of argument sources depends on other options specified, see below.

`-U` If a project file is specified and no argument source is explicitly specified (either directly or by means of `-files` option), process all the units of the closure of the argument project. Otherwise this option has no effect.

`-U main_unit` If a project file is specified and no argument source is explicitly specified (either directly or by means of `-files` option), process the closure of units rooted at *main_unit*. Otherwise this option has no effect.

`-Xname=value` Indicates that external variable *name* in the argument project has the *value* value. Has no effect if no project is specified as tool argument.

`--subdirs=dir` Use the specified subdirectory of the project objects file (or of the project file directory if the project does not specify an object directory) for tool output files. Has no effect if no project is specified as tool argument or if `--no_objects_dir` is specified.

`--no_objects_dir` Place all the result files into the current directory instead of project objects directory.

`--RTS=rts-path` Specifies the default location of the runtime library. Same meaning as the equivalent *gnatmake* flag (see GNAT User's Guide).

`-a` Process all units including those with read-only ALI files such as those from the GNAT Run-Time library.

`--incremental` Incremental processing on a per-file basis. Source files are only processed if they have been modified, or if files they depend on have been modified. This is similar to the way gnatmake/gprbuild only compiles files that need to be recompiled. A project file is required in this mode, and the gnat driver (as in *gnat check*) is not supported. Note that rules requiring a global analysis (Recursive_Subprograms, Deeply_Nested_Inlining) are not supported in –incremental mode.

`-h` List all the rules checked by the given *gnatcheck* version.

`-jnnnn` Use *nnnn* processes to carry out the tree creations (internal representations of the argument sources). On a multiprocessor machine this speeds up processing of big sets of argument sources. If *n* is 0, then the maximum number of parallel tree creations is the number of core processors on the platform.

`-l` Use full source locations references in the report file. For a construct from a generic instantiation a full source location is a chain from the location of this construct in the generic unit to the place where this unit is instantiated.

`-log` Duplicate all the output sent to `stderr` into a log file. The log file is named `gnatcheck.log`. If a project file is specified as *gnatcheck* parameter then it is located in the project objects directory (or in the project file directory if no object directory is specified). Otherwise it is located in the current directory.

-mnnnn Maximum number of diagnostics to be sent to `stdout`, where *nnnn* is in the range 0...1000; the default value is 500. Zero means that there is no limitation on the number of diagnostic messages to be output.

`-q` Quiet mode. All the diagnostics about rule violations are placed in the *gnatcheck* report file only, without duplication on `stdout`.

`-s` Short format of the report file (no version information, no list of applied rules, no list of checked sources is included)

`-xml` Generate the report file in XML format. Is not allowed in incremental mode.

`-nt` Do not generate the report file in text format. Enforces `-xml`, is not allowed in incremental mode.

`-files=filename`

Take the argument source files from the specified file. This file should be an ordinary text file containing file names separated by spaces or line breaks. You can use this switch more than once in the same call to *gnatcheck*. You also can combine this switch with an explicit list of files.

`--ignore=filename`

Do not process the sources listed in a specified file. This option cannot be used in incremental mode.

`--show-rule` Add the corresponding rule name to the diagnosis generated for its violation.

`--check-redefinition` For a parametrized rule check if a rule parameter is defined more than once in the set of rule options specified and issue a warning if parameter redefinition is detected

`--include-file=file` Append the content of the specified text file to the report file

`-t` Print out execution time.

`-v` Verbose mode; *gnatcheck* generates version information and then a trace of sources being processed.

`-o report_file` Set name of the text report file to *report_file*.

`-ox report_file` Set name of the XML report file to *report_file*. Enforces `-xml`, is not allowed in incremental mode.

`--write-rules=template_file` Write to *template_file* the template rule file that contains all the rules currently implemented in *gnatcheck* turned off. A user may edit this template file manually to get his own coding standard file.

If a project file is specified and no argument source is explicitly specified (either directly or by means of `-files` option), and no `-U` is specified, then the set of processed sources is all the immediate units of the argument project.

If the argument project file is defines aggregate project, and it aggregates more than one (non-aggregate) project, gnatcheck runs separately for each (non-aggregate) project being aggregated by the argument project, and a separate report file is created for each of these runs. Also such a run creates an umbrella report file that lists all the (non-aggregate) projects that are processed separately and for each of these projects contains the reference for the corresponding report file.

If the argument project file defines an aggregate project but it aggregates only one (non-aggregate) project, the gnatcheck behavior is the same as for the case of non-aggregate argument project file.

CHAPTER

FIVE

GNATCHECK RULE OPTIONS

The following options control the processing performed by *gnatcheck*.

`+R[:rule_synonym:]rule_id[:param{,param}]` Turn on the check for a specified rule with the specified parameter(s), if any. *rule_id* must be the identifier of one of the currently implemented rules (use `-h` for the list of implemented rules). Rule identifiers are not case-sensitive. Each *param* item must be a non-empty string representing a valid parameter for the specified rule. If the part of the rule option that follows the colon character contains any space characters then this part must be enclosed in quotation marks.

rule_synonym is a user-defined synonym for a rule name, it can be used to map *gnatcheck* rules onto a user coding standard.

`-Rrule_id[:param]` Turn off the check for a specified rule with the specified parameter, if any.

`-from=rule_option_filename` Read the rule options from the text file *rule_option_filename*, referred to as a 'coding standard file' below.

The default behavior is that all the rule checks are disabled.

If a rule option is given in a rule file, it can contain spaces and line breaks. Otherwise there should be no spaces between the components of a rule option.

If more than one rule option is specified for the same rule, these options are summed together. If a new option contradicts the rule settings specified by previous options for this rule, the new option overrides the previous settings.

A coding standard file is a text file that contains a set of rule options described above.

The file may contain empty lines and Ada-style comments (comment lines and end-of-line comments). There can be several rule options on a single line (separated by a space).

A coding standard file may reference other coding standard files by including more `-from=rule_option_filename` options, each such option being replaced with the content of the corresponding coding standard file during processing. In case a cycle is detected (that is, `rule_file_1` reads rule options from `rule_file_2`, and `rule_file_2` reads (directly or indirectly) rule options from `rule_file_1`), processing fails with an error message.

If the name of the coding standard file does not contain a path information in absolute form, then it is treated as being relative to the current directory if gnatcheck is called without a project file or as being relative to the project file directory if gnatcheck is called with a project file as an argument.

This page is intentionally left blank.

CHAPTER

SIX

ADDING THE RESULTS OF COMPILER CHECKS TO *GNATCHECK* OUTPUT

The *gnatcheck* tool can include in the generated diagnostic messages and in the report file the results of the checks performed by the compiler. Though disabled by default, this effect may be obtained by using `+R` with the following rule identifiers and parameters:

Restrictions To record restrictions violations (which are performed by the compiler if the pragma `Restrictions` or `Restriction_Warnings` are given), use the `Restrictions` rule with the same parameters as pragma `Restrictions` or `Restriction_Warnings`.

This rule allows parametric rule exemptions, the parameters that are allowed in the definition of exemption sections are the names of the restrictions except for the case when a restriction requires a non-numeric parameter, in this case the parameter should be the name of the restriction with the parameter, as it is given for the rule.

Style_Checks To record compiler style checks (see Style Checking section in GNAT User's Guide), use the `Style_Checks` rule.

This rule takes a parameter in one of the following forms:

- ***All_Checks*,** which enables the standard style checks corresponding to the `-gnatyy` GNAT style check option, or
- a string with the same structure and semantics as the `string_LITERAL` parameter of the GNAT pragma `Style_Checks` (see "Pragma Style_Checks" in the GNAT Reference Manual).

For example, the `+RStyle_Checks:O` rule option activates the compiler style check that corresponds to `-gnatyO` style check option.

Warnings To record compiler warnings (see Warning Message Control section in GNAT User's Guide), use the `Warnings` rule with a parameter that is a valid *static_string_expression* argument of the GNAT pragma `Warnings` (see "Pragma Warnings" in the GNAT Reference Manual). Note that in case of gnatcheck 's' parameter, that corresponds to the GNAT `-gnatws` option, disables all the specific warnings, but not suppresses the warning mode, and 'e' parameter, corresponding to `-gnatwe` that means "treat warnings as errors", does not have any effect.

This rule allows parametric rule exemptions, the parameters that are allowed in the definition of exemption sections are the same as the parameters of the rule itself. Note that parametric exemption sections have their effect only if either `.d` parameter is specified for the `Warnings` rule or if the `--show-rules` option is set.

To disable a specific restriction check, use `-RRestrictions` gnatcheck option with the corresponding restriction name as a parameter. `-R` is not available for `Style_Checks` and `Warnings` options, to disable warnings and style checks, use the corresponding warning and style options.

This page is intentionally left blank.

CHAPTER

SEVEN

MAPPING *GNATCHECK* RULES ONTO CODING STANDARDS

If a user would like use *gnatcheck* to check if some code satisfies to a given coding standard, the following approach can be used to simplify mapping of the coding standard requirements onto *gnatcheck* rules:

- when specifying rule options, use synonyms for the rule names that are relevant to your coding standard:

```
+R :My_Coding_Rule_1: Gnatcheck_Rule_1: param1
...
+R :My_Coding_Rule_N: Gnatcheck_Rule_N
```

- call *gnatcheck* with *–show-rule* option that adds the rule names to the generated diagnoses. If a synonym is used in the rule option that enables the rule, then this synonym will be used to annotate the diagnosis instead of the rule name:

```
foo.adb:2:28: something is wrong here [My_Coding_Rule_1]
...
bar.ads:17:3: this is not good [My_Coding_Rule_N]
```

Currently this approach does not work for compiler-based checks integrated in *gnatcheck* (implemented by *Restrictions*, *Style_Checks* and *Warnings* rules.

This page is intentionally left blank.

CHAPTER EIGHT

RULE EXEMPTION

One of the most useful applications of *gnatcheck* is to automate the enforcement of project-specific coding standards, for example in safety-critical systems where particular features must be restricted in order to simplify the certification effort. However, it may sometimes be appropriate to violate a coding standard rule, and in such cases the rationale for the violation should be provided in the source program itself so that the individuals reviewing or maintaining the program can immediately understand the intent.

The *gnatcheck* tool supports this practice with the notion of a 'rule exemption' covering a specific source code section. Normally rule violation messages are issued both on `stderr` and in a report file. In contrast, exempted violations are not listed on `stderr`; thus users invoking *gnatcheck* interactively (e.g. in its GPS interface) do not need to pay attention to known and justified violations. However, exempted violations along with their justification are documented in a special section of the report file that *gnatcheck* generates.

8.1 Using pragma `Annotate` to Control Rule Exemption

Rule exemption is controlled by pragma `Annotate` when its first argument is 'gnatcheck'. The syntax of *gnatcheck*'s exemption control annotations is as follows:

```
pragma Annotate (gnatcheck, exemption_control, Rule_Name [, justification]);

exemption_control ::= Exempt_On | Exempt_Off

Rule_Name         ::= string_literal

justification     ::= string_literal
```

When a *gnatcheck* annotation has more than four arguments, *gnatcheck* issues a warning and ignores the additional arguments. If the arguments do not follow the syntax above, *gnatcheck* emits a warning and ignores the annotation.

The `Rule_Name` argument should be the name of some existing *gnatcheck* rule. Otherwise a warning message is generated and the pragma is ignored. If `Rule_Name` denotes a rule that is not activated by the given *gnatcheck* call, the pragma is ignored and no warning is issued. The exception from this rule is that exemption sections for `Warnings` rule are fully processed when `Restrictions` rule is activated.

A source code section where an exemption is active for a given rule is delimited by an `exempt_on` and `exempt_off` annotation pair:

```
pragma Annotate (gnatcheck, Exempt_On, "Rule_Name", "justification");
-- source code section
pragma Annotate (gnatcheck, Exempt_Off, "Rule_Name");
```

For some rules it is possible specify rule parameter(s) when defining an exemption section for a rule. This means that only the checks corresponding to the given rule parameter(s) are exempted in this section:

```
pragma Annotate (gnatcheck, Exempt_On, "Rule_Name: Par1, Par2", "justification");
-- source code section
pragma Annotate (gnatcheck, Exempt_Off, "Rule_Name: Par1, Par2");
```

A parametric exemption section can be defined for a rule if a rule has parameters and these parameters change the scope of the checks performed by a rule. For example, if you define an exemption section for 'Restriction' rule with the parameter 'No_Allocators', then in this section only the checks for `No_Allocators` will be exempted, and the checks for all the other restrictions from your coding standard will be performed as usial.

See the description of individual rules to check if parametric exemptions are available for them and what is the format of the rule parameters to be used in the corresponding parameters of the `Annotate` pragmas.

8.2 *gnatcheck* Annotations Rules

- An 'Exempt_Off' annotation can only appear after a corresponding 'Exempt_On' annotation.
- Exempted source code sections are only based on the source location of the annotations. Any source construct between the two annotations is part of the exempted source code section.
- Exempted source code sections for different rules are independent. They can be nested or intersect with one another without limitation. Creating nested or intersecting source code sections for the same rule is not allowed.
- A matching 'Exempt_Off' annotation pragma for an 'Exempt_On' pragma that defines a parametric exemption section is the pragma that contains exactly the same set of rule parameters for the same rule.
- Parametric exemption sections for the same rule with different parameters can intersect or overlap in case if the parameter sets for such sections have an empty intersection.
- Malformed exempted source code sections are reported by a warning, and the corresponding rule exemptions are ignored.
- When an exempted source code section does not contain at least one violation of the exempted rule, a warning is emitted on `stderr`.
- If an 'Exempt_On' annotation pragma does not have a matching 'Exempt_Off' annotation pragma in the same compilation unit, a warning is issued and the exemption section is considered to last until the end of the compilation unit source.

CHAPTER

NINE

PREDEFINED RULES

The description of the rules currently implemented in *gnatcheck* is given in this chapter. The rule identifier is used as a parameter of *gnatcheck*'s `+R` or `-R` switches.

Be aware that most of these rules apply to specialized coding requirements developed by individual users and may well not make sense in other environments. In particular, there are many rules that conflict with one another. Proper usage of gnatcheck involves selecting the rules you wish to apply by looking at your independently developed coding standards and finding the corresponding gnatcheck rules.

If not otherwise specified, a rule does not do any check for the results of generic instantiations.

9.1 Style-Related Rules

The rules in this section may be used to enforce various feature usages consistent with good software engineering, for example as described in Ada 95 Quality and Style.

9.1.1 Tasking

The rules in this subsection may be used to enforce various feature usages related to concurrency.

`Multiple_Entries_In_Protected_Definitions`

Flag each protected definition (i.e., each protected object/type declaration) that declares more than one entry. Diagnostic messages are generated for all the entry declarations except the first one. An entry family is counted as one entry. Entries from the private part of the protected definition are also checked.

This rule has no parameters.

Example

```
protected PO is
   entry Get (I :     Integer);
   entry Put (I : out Integer);    --  FLAG
   procedure Reset;
   function Check return Boolean;
private
   Val : Integer := 0;
end PO;
```

Volatile_Objects_Without_Address_Clauses

Flag each volatile object that does not have an address specification. Only variable declarations are checked.

An object is considered as being volatile if a pragma or aspect Volatile is applied to the object or to its type, if the object is atomic or if the GNAT compiler considers this object as volatile because of some code generation reasons.

This rule has no parameters.

Example

```
with Interfaces, System, System.Storage_Elements;
package Foo is
   Variable: Interfaces.Unsigned_8
      with Address => System.Storage_Elements.To_Address (0), Volatile;

   Variable1: Interfaces.Unsigned_8                                --  FLAG
      with Volatile;

   type My_Int is range 1 .. 32 with Volatile;

   Variable3 : My_Int;                                             --  FLAG

   Variable4 : My_Int
     with Address => Variable3'Address;
end Foo;
```

9.1.2 Object Orientation

The rules in this subsection may be used to enforce various feature usages related to Object-Oriented Programming.

Constructors

Flag any declaration of a primitive function of a tagged type that has a controlling result and no controlling parameter. If a declaration is a completion of another declaration then it is not flagged.

This rule has no parameters.

Example

```
type T is tagged record
   I : Integer;
end record;

function Fun (I : Integer) return T;                 -- FLAG
function Bar (J : Integer) return T renames Fun;     -- FLAG
function Foo (K : Integer) return T is ((I => K));   -- FLAG
```

`Deep_Inheritance_Hierarchies`

Flags a tagged derived type declaration or an interface type declaration if its depth (in its inheritance hierarchy) exceeds the value specified by the *N* rule parameter. Types in generic instantiations which violate this rule are also flagged; generic formal types are not flagged. This rule also does not flag private extension declarations. In the case of a private extension, the corresponding full declaration is checked.

In most cases, the inheritance depth of a tagged type or interface type is defined as 0 for a type with no parent and no progenitor, and otherwise as 1 + max of the depths of the immediate parent and immediate progenitors. If the declaration of a formal derived type has no progenitor, or if the declaration of a formal interface type has exactly one progenitor, then the inheritance depth of such a formal derived/interface type is equal to the inheritance depth of its parent/progenitor type, otherwise the general rule is applied.

If the rule flags a type declaration inside the generic unit, this means that this type declaration will be flagged in any instantiation of the generic unit. But if a type is derived from a format type or has a formal progenitor and it is not flagged at the place where it is defined in a generic unit, it may or may not be flagged in instantiation, this depends of the inheritance depth of the actual parameters.

This rule has the following (mandatory) parameter for the `+R` option:

N Integer not less than -1 specifying the maximal allowed depth of any inheritance hierarchy. If the rule parameter is set to -1, the rule flags all the declarations of tagged and interface types.

Example

```
type I0 is interface;
type I1 is interface and I0;
type I2 is interface and I1;

type T0 is tagged null record;
type T1 is new T0 and I0 with null record;
type T2 is new T0 and I1 with null record;
type T3 is new T0 and I2 with null record; -- FLAG (if rule parameter is 2)
```

`Direct_Calls_To_Primitives`

Flag any non-dispatching call to a dispatching primitive operation, except for:

- a call to the corresponding primitive of the parent type. (This occurs in the common idiom where a primitive subprogram for a tagged type directly calls the same primitive subprogram of the parent type.)
- a call to a primitive of an untagged private type, even though the full type may be tagged, when the call is made at a place where the view of the type is untagged.

This rule has the following (optional) parameters for the `+R` option:

Except_Constructors Do not flag non-dispatching calls to functions if the function has a controlling result and no controlling parameters (in a traditional OO sense such functions may be considered as constructors).

Example

```
package Root is
   type T_Root is tagged private;
   procedure Primitive_1 (X : in out T_Root);
   procedure Primitive_2 (X : in out T_Root);
private
   type T_Root is tagged record
      Comp : Integer;
   end record;
end Root;

package Root.Child is
   type T_Child is new T_Root with private;

   procedure Primitive_1 (X : in out T_Child);
   procedure Primitive_2 (X : in out T_Child);
private
   type T_Child is new T_Root with record
      B : Boolean;
   end record;
end Root.Child;

package body Root.Child is

   procedure Primitive_1 (X : in out T_Child) is
   begin
      Primitive_1 (T_Root (X));       --  NO FLAG
      Primitive_2 (T_Root (X));       --  FLAG
      Primitive_2 (X);                --  FLAG
   end Primitive_1;

   procedure Primitive_2 (X : in out T_Child) is
   begin
      X.Comp  := X.Comp + 1;
   end Primitive_2;

end Root.Child;
```

Downward_View_Conversions

Flag downward view conversions.

This rule has no parameters.

Example

```
package Foo is
   type T1 is tagged private;
   procedure Proc1 (X : in out T1'Class);

   type T2 is new T1 with private;
   procedure Proc2 (X : in out T2'Class);

private
   type T1 is tagged record
      C : Integer := 0;
   end record;
   type T2 is new T1 with null record;
end Foo;

package body Foo is

   procedure Proc1 (X : in out T1'Class) is
      Var : T2 := T2 (X);                   --  FLAG
   begin
      Proc2 (T2'Class (X));                 --  FLAG
   end Proc1;

   procedure Proc2 (X : in out T2'Class) is
   begin
      X.C := X.C + 1;
   end Proc2;

end Foo;
```

No_Inherited_Classwide_Pre

Flag a declaration of an overriding primitive operation of a tagged type if at least one of the operations it overrides or implements does not have (explicitly defined or inherited) Pre'Class aspect defined for it.

This rule has no parameters.

Example

```
package Foo is

   type Int is interface;
   function Test (X : Int) return Boolean is abstract;
   procedure Proc (I : in out Int) is abstract with Pre'Class => Test (I);

   type Int1 is interface;
   procedure Proc (I : in out Int1) is abstract;

   type T is tagged private;

    type NT1 is new T and Int with private;
    function Test (X : NT1) return Boolean;         --  FLAG
    procedure Proc (X : in out NT1);

    type NT2 is new T and Int1 with private;
    procedure Proc (X : in out NT2);                --  FLAG

   private
   type T is tagged record
      I : Integer;
   end record;

   type NT1 is new T and Int with null record;
   type NT2 is new T and Int1 with null record;

end Foo;
```

Specific_Pre_Post

Flag a declaration of a primitive operation of a tagged type if this declaration contains specification of Pre or/and Post aspect.

This rule has no parameters.

Example

```
type T is tagged private;
function Check1 (X : T) return Boolean;
function Check2 (X : T) return Boolean;

procedure Proc1 (X : in out T)            --  FLAG
   with Pre => Check1 (X);

procedure Proc2 (X : in out T)            --  FLAG
   with Post => Check2 (X);

function Fun1 (X : T) return Integer      --  FLAG
   with Pre  => Check1 (X),
        Post => Check2 (X);

function Fun2 (X : T) return Integer
   with Pre'Class  => Check1 (X),
        Post'Class => Check2 (X);

function Fun3 (X : T) return Integer      --  FLAG
   with Pre'Class  => Check1 (X),
        Post'Class => Check2 (X),
        Pre        => Check1 (X),
        Post       => Check2 (X);
```

Specific_Parent_Type_Invariant

Flag any record extension definition or private extension definition if a parent type has a Type_Invariant aspect defined for it. A record extension definition is not flagged if it is a part of a completion of a private extension declaration.

This rule has no parameters.

Example

```
package Pack1 is
   type PT1 is tagged private;
   type PT2 is tagged private
     with Type_Invariant => Invariant_2 (PT2);

   function Invariant_2   (X : PT2) return Boolean;

private
   type PT1 is tagged record
      I : Integer;
   end record;
   type PT2 is tagged record
      I : Integer;
   end record;

   type PT1_N is new PT1 with null record;
   type PT2_N is new PT2 with null record;     --  FLAG
end Pack1;

package Pack2 is
   type N_PT1 is new Pack1.PT1 with private;
   type N_PT2 is new Pack1.PT2 with private;  --  FLAG
private
   type N_PT1 is new Pack1.PT1 with null record;
   type N_PT2 is new Pack1.PT2 with null record;
end Pack2;
```

Specific_Type_Invariants

Flag any definition of (non-class-wide) Type_Invariant aspect that is a part of a declaration of a tagged type or a tagged extension. Definitions of Type_Invariant'Class aspects are not flagged. Definitions of (non-class-wide) Type_Invariant aspect that are parts of declarations of non-tagged types are not flagged.

This rule has no parameters.

Example

```
type PT is private
   with Type_Invariant => Test_PT (PT);
function Test_PT (X : PT) return Boolean;

type TPT1 is tagged private
   with Type_Invariant => Test_TPT1 (TPT1);        --  FLAG
function Test_TPT1 (X : TPT1) return Boolean;

type TPT2 is tagged private
   with Type_Invariant'Class => Test_TPT2 (TPT2);
function Test_TPT2 (X : TPT2) return Boolean;
```

Too_Many_Parents

Flag any tagged type declaration, interface type declaration, single task declaration or single protected declaration that has more than *N parents*, where *N* is a parameter of the rule. A *parent* here is either a (sub)type denoted by the subtype mark from the parent_subtype_indication (in case of a derived type declaration), or any of the progenitors from the interface list (if any).

This rule has the following (mandatory) parameters for the `+R` option:

N Positive integer specifying the maximal allowed number of parents/progenitors.

Example

```
type I1 is interface;
type I2 is interface;
type I3 is interface;
type I4 is interface;

type T_Root is tagged private;

type T_1 is new T_Root with private;
type T_2 is new T_Root and I1 with private;
type T_3 is new T_Root and I1 and I2 with private;
type T_4 is new T_Root and I1 and I2 and I3 with private; -- FLAG (if rule parameter is 3 or less)
```

`Too_Many_Primitives`

Flag any tagged type declaration that has more than N user-defined primitive operations (counting both inherited and not overridden and explicitly declared, not counting predefined operators). Do not flag type declarations that are completions of private type or extension declarations.

This rule has the following (mandatory) parameters for the `+R` option:

N Positive integer specifying the maximal number of primitives when the type is not flagged.

Example

```
package Foo is
   type PT is tagged private;     --  FLAG (if rule parameter is 3 or less)

   procedure P1 (X : in out PT);
   procedure P2 (X : in out PT) is null;
   function F1 (X : PT) return Integer;
   function F2 (X : PT) return Integer is (F1 (X) + 1);

   type I1 is interface;

   procedure P1 (X : in out I1) is abstract;
   procedure P2 (X : in out I1) is null;

   type I2 is interface and I1;   --  FLAG (if rule parameter is 3 or less)
   function F1 (X : I2) return Integer is abstract;
   function F2 (X : I2) return Integer is abstract;

private
   type PT is tagged record
      I : Integer;
   end record;
end Foo;
```

`Visible_Components`

Flag all the type declarations located in the visible part of a library package or a library generic package that can declare a visible component. A visible component can be declared in a *record definition* which appears on its own or as part of a record extension. The *record definition* is flagged even if it contains no components.

Record definitions located in private parts of library (generic) packages or in local (generic) packages are not flagged. *Record definitions* in private packages, in package bodies, and in the main subprogram body are not flagged.

This rule has the following (optional) parameters for the `+R` option:

Tagged_Only Only declarations of tagged types are flagged.

Example

```
with Types;
package Foo is
   type Null_Record is null record;                                   --  FLAG

   type Not_Null_Record is record                                     --  FLAG
      I : Integer;
      B : Boolean;
   end record;

   type Tagged_Not_Null_Record is tagged record                       --  FLAG
      I : Integer;
      B : Boolean;
   end record;

   type Private_Extension is new Types.Tagged_Private with private;

   type NoN_Private_Extension is new Types.Tagged_Private with record  --  FLAG
      B : Boolean;
   end record;

private
   type Rec is tagged record
      I : Integer;
   end record;

   type Private_Extension is new Types.Tagged_Private with record
      C : Rec;
   end record;
end Foo;
```

9.1.3 Portability

The rules in this subsection may be used to enforce various feature usages that support program portability.

`Bit_Records_Without_Layout_Definition`

Flag record type declarations if a record has a component of a modular type and the record type does not have a record representation clause applied to it.

This rule has no parameters.

Example

```
package Pack is
   type My_Mod is mod 8;

   type My_Rec is record   --  FLAG
      I : My_Mod;
   end record;
end Pack;
```

`Forbidden_Attributes`

Flag each use of the specified attributes. The attributes to be detected are named in the rule's parameters.

This rule has the following parameters:

- For the `+R` option

Attribute_Designator Adds the specified attribute to the set of attributes to be detected and sets the detection checks for all the specified attributes ON. If *Attribute_Designator* does not denote any attribute defined in the Ada standard or in the GNAT Reference Manual, it is treated as the name of unknown attribute.

`GNAT` All the GNAT-specific attributes are detected; this sets the detection checks for all the specified attributes ON.

`ALL` All attributes are detected; this sets the rule ON.

- For the `-R` option

Attribute_Designator Removes the specified attribute from the set of attributes to be detected without affecting detection checks for other attributes. If *Attribute_Designator* does not correspond to any attribute defined in the Ada standard or in the GNAT Reference Manual, this option is treated as turning OFF detection of all unknown attributes.

GNAT Turn OFF detection of all GNAT-specific attributes

ALL Clear the list of the attributes to be detected and turn the rule OFF.

Parameters are not case sensitive. If *Attribute_Designator* does not have the syntax of an Ada identifier and therefore can not be considered as a (part of an) attribute designator, a diagnostic message is generated and the corresponding parameter is ignored. (If an attribute allows a static expression to be a part of the attribute designator, this expression is ignored by this rule.)

When more than one parameter is given in the same rule option, the parameters must be separated by commas.

If more than one option for this rule is specified for the gnatcheck call, a new option overrides the previous one(s).

The `+R` option with no parameters turns the rule ON, with the set of attributes to be detected defined by the previous rule options. (By default this set is empty, so if the only option specified for the rule is `+RForbidden_Attributes` (with no parameter), then the rule is enabled, but it does not detect anything). The `-R` option with no parameter turns the rule OFF, but it does not affect the set of attributes to be detected.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are *Attribute_Designators*. Each *Attribute_Designator* used as a rule exemption parameter should denote a predefined or GNAT-specific attribute.

Example

```
--  if the rule is activated as +RForbidden_Attributes:Range,First,Last
procedure Foo is
   type Arr is array (1 .. 10) of Integer;
   Arr_Var : Arr;

   subtype Ind is Integer range Arr'First .. Arr'Last; --  FLAG (twice)
begin

   for J in Arr'Range loop                             --  FLAG
      Arr_Var (J) := Integer'Succ (J);
```

`Forbidden_Pragmas`

Flag each use of the specified pragmas. The pragmas to be detected are named in the rule's parameters.

This rule has the following parameters:

- For the `+R` option

Pragma_Name Adds the specified pragma to the set of pragmas to be checked and sets the checks for all the specified pragmas ON. *Pragma_Name* is treated as a name of a pragma. If it does not correspond to any pragma name defined in the Ada standard or to the name of a GNAT-specific pragma defined in the GNAT Reference Manual, it is treated as the name of unknown pragma.

`GNAT` All the GNAT-specific pragmas are detected; this sets the checks for all the specified pragmas ON.

`ALL` All pragmas are detected; this sets the rule ON.

- For the `-R` option

Pragma_Name Removes the specified pragma from the set of pragmas to be checked without affecting checks for other pragmas. *Pragma_Name* is treated as a name of a pragma. If it does not correspond to any pragma defined in the Ada standard or to any name defined in the GNAT Reference Manual, this option is treated as turning OFF detection of all unknown pragmas.

GNAT Turn OFF detection of all GNAT-specific pragmas

ALL Clear the list of the pragmas to be detected and turn the rule OFF.

Parameters are not case sensitive. If *Pragma_Name* does not have the syntax of an Ada identifier and therefore can not be considered as a pragma name, a diagnostic message is generated and the corresponding parameter is ignored.

When more than one parameter is given in the same rule option, the parameters must be separated by a comma.

If more than one option for this rule is specified for the *gnatcheck* call, a new option overrides the previous one(s).

The `+R` option with no parameters turns the rule ON with the set of pragmas to be detected defined by the previous rule options. (By default this set is empty, so if the only option specified for the rule is `+RForbidden_Pragmas` (with no parameter), then the rule is enabled, but it does not detect anything). The `-R` option with no parameter turns the rule OFF, but it does not affect the set of pragmas to be detected.

Note that in case when the rule is enabled with *ALL* parameter, then the rule will flag also pragmas `Annotate` used to exempt rules, see *Rule exemption*. Even if you exempt this *Forbidden_Pragmas* rule then the pragma `Annotate` that closes the exemption section will be flagged as non-exempted. To avoid this, turn off the check for pragma `Annotate` by using `-RForbidden_Pragmas:Annotate` rule option.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are pragma names. Each name used as a rule exemption parameter should denote a predefined or GNAT-specific pragma.

Example

```
--  if the rule is activated as +RForbidden_Pragmas:Pack
package Foo is

   type Arr is array (1 .. 8) of Boolean;
   pragma Pack (Arr);                      --  FLAG

   I : Integer;
   pragma Atomic (I);

end Foo;
```

`Implicit_SMALL_For_Fixed_Point_Types`

Flag each fixed point type declaration that lacks an explicit representation clause to define its `'Small` value. Since `'Small` can be defined only for ordinary fixed point types, decimal fixed point type declarations are not checked.

This rule has no parameters.

Example

```
package Foo is
   type Fraction is delta 0.01 range -1.0 .. 1.0;
   type Fraction1 is delta 0.01 range -1.0 .. 1.0; --  FLAG

   type Money is delta 0.01 digits 15;

   for Fraction'Small use 0.01;
end Foo;
```

`Incomplete_Representation_Specifications`

Flag all record types that have a layout representation specification but without Size and Pack representation specifications.

This rule has no parameters.

Example

```
package Pack is
   type Rec is record  --  FLAG
      I : Integer;
      B : Boolean;
   end record;

   for Rec use record
      I at 0 range 0 ..31;
      B at 4 range 0 .. 7;
   end record;
end Pack;
```

No_Explicit_Real_Range

Flag a declaration of a floating point type or a decimal fixed point type, including types derived from them if no explicit range specification is provided for the type.

This rule has no parameters.

Example

```
type F1 is digits 8;                              --  FLAG
type F2 is delta 0.01 digits 8;                   --  FLAG
```

No_Scalar_Storage_Order_Specified

Flag each record type declaration, record extension declaration, and untagged derived record type declaration if a record_representation_clause that has at least one component clause applies to it (or an ancestor), but neither the type nor any of its ancestors has an explicitly specified Scalar_Storage_Order attribute.

This rule has no parameters.

Example

```
with System;
package Foo is

   type Rec1 is  record      --  FLAG
      I : Integer;
   end record;

   for Rec1 use
      record
         I at 0 range 0 .. 31;
      end record;

   type Rec2 is  record
      I : Integer;
   end record;

   for Rec2 use
      record
         I at 0 range 0 .. 31;
      end record;

   pragma Attribute_Definition (Scalar_Storage_Order, Rec2, System.Low_Order_First);
end Foo;
```

`Predefined_Numeric_Types`

Flag each explicit use of the name of any numeric type or subtype declared in package `Standard`.

The rationale for this rule is to detect when the program may depend on platform-specific characteristics of the implementation of the predefined numeric types. Note that this rule is overly pessimistic; for example, a program that uses `String` indexing likely needs a variable of type `Integer`. Another example is the flagging of predefined numeric types with explicit constraints:

```
subtype My_Integer is Integer range Left .. Right;
Vy_Var : My_Integer;
```

This rule detects only numeric types and subtypes declared in package `Standard`. The use of numeric types and subtypes declared in other predefined packages (such as `System.Any_Priority` or `Ada.Text_IO.Count`) is not flagged

This rule has no parameters.

Example

```
package Foo is
   I : Integer;                               -- FLAG
   F : Float;                                 -- FLAG
   B : Boolean;

   type Arr is array (1 .. 5) of Short_Float; -- FLAG

   type Res is record
      C1 : Long_Integer;                      -- FLAG
      C2 : Character;
   end record;

end Foo;
```

`Printable_ASCII`

Flag source code text characters that are not part of the printable ASCII character set, a line feed, or a carriage return character (i.e. values 10, 13 and 32 .. 126 of the ASCII Character set).

If a code line contains more than one symbol that does not belong to the printable ASCII character set, the generated diagnosis points to the first (leftmost) character and says that there are more in this line.

This rule has no parameters.

`Separate_Numeric_Error_Handlers`

Flags each exception handler that contains a choice for the predefined `Constraint_Error` exception, but does not contain the choice for the predefined `Numeric_Error` exception, or that contains the choice for `Numeric_Error`, but does not contain the choice for `Constraint_Error`.

This rule has no parameters.

Example

```
exception
   when Constraint_Error =>  --  FLAG
      Clean_Up;
end;
```

9.1.4 Program Structure

The rules in this subsection may be used to enforce feature usages related to program structure.

`Deep_Library_Hierarchy`

Flag any library package declaration, library generic package declaration or library package instantiation that has more than N parents and grandparents (that is, the name of such a library unit contains more than N dots). Child subprograms, generic subprograms subprogram instantiations and package bodies are not flagged.

This rule has the following (mandatory) parameters for the `+R` option:

N Positive integer specifying the maximal number of ancestors when the unit is not flagged.

Example

```
package Parent.Child1.Child2 is  -- FLAG  (if rule parameter is 1)
   I : Integer;
end;
```

`Deeply_Nested_Generics`

Flag a generic declaration nested in another generic declaration if the nesting level of the inner generic exceeds the value specified by the *N* rule parameter. The nesting level is the number of generic declarations that enclose the given (generic) declaration. Formal packages are not flagged by this rule.

This rule has the following (mandatory) parameters for the `+R` option:

N Nonnegative integer specifying the maximum nesting level for a generic declaration.

Example

```
package Foo is

   generic
   package P_G_0 is
      generic
      package P_G_1 is
         generic              --  FLAG (if rule parameter is 1)
         package P_G_2 is
            I  : Integer;
         end;
      end;
   end;

end Foo;
```

Local_Packages

Flag all local packages declared in package and generic package specs. Local packages in bodies are not flagged.

This rule has no parameters.

Example

```
package Foo is
   package Inner is    --  FLAG
      I : Integer;
   end Inner;
end Foo;
```

Non_Visible_Exceptions

Flag constructs leading to the possibility of propagating an exception out of the scope in which the exception is declared. Two cases are detected:

- An exception declaration in a subprogram body, task body or block statement is flagged if the body or statement does not contain a handler for that exception or a handler with an `others` choice.
- A `raise` statement in an exception handler of a subprogram body, task body or block statement is flagged if it (re)raises a locally declared exception. This may occur under the following circumstances:
  - it explicitly raises a locally declared exception, or
  - it does not specify an exception name (i.e., it is simply `raise;`) and the enclosing handler contains a locally declared exception in its exception choices.

Renamings of local exceptions are not flagged.

This rule has no parameters.

Example

```
procedure Bar is
   Var : Integer :=- 13;

   procedure Inner (I : in out Integer) is
      Inner_Exception_1 : exception;          --  FLAG
      Inner_Exception_2 : exception;
   begin
      if I = 0 then
         raise Inner_Exception_1;
      elsif I = 1 then
         raise Inner_Exception_2;
      else
         I := I - 1;
      end if;
   exception
      when Inner_Exception_2 =>
         I := 0;
         raise;                               --  FLAG
   end Inner;

begin
   Inner (Var);
end Bar;
```

`Raising_External_Exceptions`

Flag any `raise` statement, in a program unit declared in a library package or in a generic library package, for an exception that is neither a predefined exception nor an exception that is also declared (or renamed) in the visible part of the package.

This rule has no parameters.

Example

```
package Exception_Declarations is
   Ex : exception;
end Exception_Declarations;
package Foo is
   procedure Proc (I : in out Integer);
end Foo;
with Exception_Declarations;
package body Foo is
   procedure Proc (I : in out Integer) is
   begin
      if I < 0 then
         raise Exception_Declarations.Ex;   --  FLAG
      else
         I := I - 1;
      end if;
   end Proc;
end Foo;
```

9.1.5 Programming Practice

The rules in this subsection may be used to enforce feature usages that relate to program maintainability.

`Address_Specifications_For_Initialized_Objects`

Flag address clauses and address aspect definitions if they are applied to object declarations with explicit initializations.

This rule has no parameters.

Example

```
I : Integer := 0;
Var0 : Integer with Address => I'Address;

Var1 : Integer := 10;
for Var1'Address use Var0'Address;              --  FLAG
```

`Address_Specifications_For_Local_Objects`

Flag address clauses and address aspect definitions if they are applied to data objects declared in local subprogram bodies. Data objects declared in library subprogram bodies are not flagged.

This rule has no parameters.

Example

```
package Pack is
   Var : Integer;
   procedure Proc (I : in out Integer);
end Pack;
package body Pack is
   procedure Proc (I : in out Integer) is
      Tmp : Integer with Address => Pack.Var'Address;    --  FLAG
   begin
      I := Tmp;
   end Proc;
end Pack;
```

`Anonymous_Arrays`

Flag all anonymous array type definitions (by Ada semantics these can only occur in object declarations).

This rule has no parameters.

Example

```
type Arr is array (1 .. 10) of Integer;
Var1 : Arr;
Var2 : array (1 .. 10) of Integer;       --  FLAG
```

`Binary_Case_Statements`

Flag a case statement if this statement has only two alternatives, one containing exactly one choice, the other containing exactly one choice or the `OTHERS` choice.

This rule has no parameters.

Example

```
case Var is                          --  FLAG
   when 1 =>
      Var := Var + 1;
   when others =>
      null;
end case;
```

`Default_Values_For_Record_Components`

Flag a record component declaration if it contains a default expression. Do not flag record component declarations in protected definitions. Do not flag discriminant specifications.

This rule has no parameters.

Example

```
type Rec (D : Natural := 0) is record
   I : Integer := 0;                    -- FLAG
   B : Boolean;

   case D is
      when 0 =>
         C : Character := 'A';          -- FLAG
      when others =>
         F : Float;
   end case;
end record;
```

`Deriving_From_Predefined_Type`

Flag derived type declaration if the ultimate ancestor type is a predefined Ada type. Do not flag record extensions and private extensions. The rule is checked inside expanded generics.

This rule has no parameters.

Example

```
package Foo is
   type T is private;
   type My_String is new String;   --  FLAG
private
   type T is new Integer;          --  FLAG
end Foo;
```

Enumeration_Ranges_In_CASE_Statements

Flag each use of a range of enumeration literals as a choice in a `case` statement. All forms for specifying a range (explicit ranges such as `A .. B`, subtype marks and `'Range` attributes) are flagged. An enumeration range is flagged even if contains exactly one enumeration value or no values at all. A type derived from an enumeration type is considered as an enumeration type.

This rule helps prevent maintenance problems arising from adding an enumeration value to a type and having it implicitly handled by an existing `case` statement with an enumeration range that includes the new literal.

This rule has no parameters.

Example

```
procedure Bar (I : in out Integer) is
   type Enum is (A, B, C, D, E);
   type Arr is array (A .. C) of Integer;

   function F (J : Integer) return Enum is separate;
begin
   case F (I) is
      when Arr'Range  =>  --  FLAG
         I := I + 1;
      when D .. E =>      --  FLAG
         null;
   end case;
end Bar;
```

Enumeration_Representation_Clauses

Flag enumeration representation clauses.

This rule has no parameters.

Example

```
type Enum1 is (A1, B1, C1);
for Enum1 use (A1 => 1, B1 => 11, C1 => 111);     --  FLAG
```

Exceptions_As_Control_Flow

Flag each place where an exception is explicitly raised and handled in the same subprogram body. A `raise` statement in an exception handler, package body, task body or entry body is not flagged.

The rule has no parameters.

Example

```
procedure Bar (I : in out Integer) is

begin
   if I = Integer'Last then
      raise Constraint_Error;     --  FLAG
   else
     I := I - 1;
   end if;
exception
   when Constraint_Error =>
      I := Integer'First;
end Bar;
```

Exits_From_Conditional_Loops

Flag any exit statement if it transfers the control out of a `for` loop or a `while` loop. This includes cases when the `exit` statement applies to a `FOR` or `while` loop, and cases when it is enclosed in some `for` or `while` loop, but transfers the control from some outer (unconditional) `loop` statement.

The rule has no parameters.

Example

```
function Bar (S : String) return Natural is
   Result : Natural := 0;
begin
   for J in S'Range loop
      exit when S (J) = '@';  --  FLAG
      Result := Result + J;
   end loop;

   return 0;
end Bar;
```

EXIT_Statements_With_No_Loop_Name

Flag each `exit` statement that does not specify the name of the loop being exited.

The rule has no parameters.

Example

```
procedure Bar (I, J : in out Integer) is
begin
   loop
      exit when I < J;  --  FLAG
      I := I - 1;
      J := J + 1;
   end loop;
end Bar;
```

Global_Variables

Flag any variable declaration that appears immediately within the specification of a library package or library generic package. Variable declarations in nested packages and inside package instantiations are not flagged.

This rule has the following (optional) parameters for the +R option:

Only_Public Do not flag variable declarations in private library (generic) packages and in package private parts.

Example

```
package Foo is
    Var1 : Integer;     --  FLAG
    procedure Proc;
private
    Var2 : Boolean;     --  FLAG
end Foo;
```

GOTO_Statements

Flag each occurrence of a goto statement.

This rule has no parameters.

Example

```
for K in 1 .. 10 loop
   if K = 6 then
      goto Quit; -- FLAG
   end if;
   null;
end loop;
<<Quit>>
return;
```

Improper_Returns

Flag each explicit return statement in procedures, and multiple return statements in functions. Diagnostic messages are generated for all return statements in a procedure (thus each procedure must be written so that it returns implicitly at the end of its statement part), and for all return statements in a function after the first one. This rule supports the stylistic convention that each subprogram should have no more than one point of normal return.

This rule has no parameters.

Example

```
procedure Proc (I : in out Integer) is
begin
   if I = 0 then
      return;                         --  FLAG
   end if;

   I := I * (I + 1);
end Proc;

function Factorial (I : Natural) return Positive is
begin
   if I = 0 then
      return 1;
   else
      return I * Factorial (I - 1);     --  FLAG
   end if;
exception
   when Constraint_Error =>
      return Natural'Last;              --  FLAG
end Factorial;
```

Local_USE_Clauses

Use clauses that are not parts of compilation unit context clause are flagged. The rule has an optional parameter for +R option:

Except_USE_TYPE_Clauses Do not flag local use type clauses.

Example

```
with Pack1;
with Pack2;
procedure Proc is
   use Pack1;               --  FLAG

   procedure Inner is
      use type Pack2.T;     --  FLAG (if Except_USE_TYPE_Clauses is not set)
   ...
```

Maximum_Parameters

Flag any subprogram declaration, subprogram body declaration, expression function declaration, null procedure declaration, subprogram body stub or generic subprogram declaration if the corresponding subprogram has more than *N* formal parameters, where *N* is a parameter of the rule.

A subprogram body, an expression function, a null procedure or a subprogram body stub is flagged only if there is no separate declaration for this subprogram. Subprogram renaming declarations and subprogram instantiations, as well as declarations inside expanded generic instantiations are never flagged.

This rule has the following (mandatory) parameters for the `+R` option:

N Positive integer specifying the maximum allowed total number of subprogram formal parameters.

Example

```
package Foo is

   procedure Proc_1 (I : in out Integer);
   procedure Proc_2 (I, J : in out Integer);
   procedure Proc_3 (I, J, K : in out Integer);
   procedure Proc_4 (I, J, K, L : in out Integer); --  FLAG (if rule parameter is 3)

   function Fun_4                                  --  FLAG (if rule parameter is 3)
     (I : Integer;
      J : Integer;
      K : Integer;
      L : Integer) return Integer is (I + J * K - L);

end Foo;
```

Misplaced_Representation_Items

Flag a representation item if there is any Ada construct except another representation item for the same entity between this clause and the declaration of the entity it applies to. A representation item in the context of this rule is either a representation clause or one of the following representation pragmas:

- Atomic J.15.8(9/3)
- Atomic_Components J.15.8(9/3)
- Independent J.15.8(9/3)
- Independent_Components J.15.8(9/3)
- Pack J.15.3(1/3)
- Unchecked_Union J.15.6(1/3)
- Volatile J.15.8(9/3)
- Volatile_Components J.15.8(9/3)

This rule has no parameters.

Example

```
type Int1 is range 0 .. 1024;
type Int2 is range 0 .. 1024;

for Int2'Size use 16;          --  NO FLAG
for Int1'Size use 16;          --  FLAG
```

Nested_Subprograms

Flag any subprogram declaration, subprogram body declaration, subprogram instantiation, expression function declaration or subprogram body stub that is not a completion of another subprogram declaration and that is declared within subprogram body (including bodies of generic subprograms), task body or entry body directly or indirectly (that is - inside a local nested package). Protected subprograms are not flagged. Null procedure declarations are not flagged. Procedure declarations completed by null procedure declarations are not flagged.

This rule has no parameters.

Example

```
procedure Bar (I, J : in out Integer) is

   procedure Foo (K : Integer) is null;
   procedure Proc1;                    --  FLAG

   procedure Proc2 is separate;        --  FLAG

   procedure Proc1 is
   begin
      I := I + J;
   end Proc1;

begin
```

Non_Short_Circuit_Operators

Flag all calls to predefined `and` and `or` operators for any boolean type. Calls to user-defined `and` and `or` and to operators defined by renaming declarations are not flagged. Calls to predefined `and` and `or` operators for modular types or boolean array types are not flagged.

This rule has no parameters.

Example

```
B1 := I > 0 and J > 0;       --  FLAG
B2 := I < 0 and then J < 0;
B3 := I > J or J > 0;        --  FLAG
B4 := I < J or else I < 0;
```

Null_Paths

Flag a statement sequence that is a component of an IF, CASE or LOOP statement if this sequences consists of NULL statements only.

This rule has no parameters.

Example

```
if I > 10 then
   J := 5;
elsif I > 0 then
   null;                 --  FLAG
else
```

```
   J := J + 1;
end if;

case J is
   when 1 =>
      I := I + 1;
   when 2 =>
      null;               --  FLAG
   when 3 =>
      J := J + 1;
   when others =>
      null;               --  FLAG
end case;
```

`Objects_Of_Anonymous_Types`

Flag any object declaration located immediately within a package declaration or a package body (including generic packages) if it uses anonymous access or array type definition. Record component definitions and parameter specifications are not flagged. Formal object declarations defined with anonymous access definitions are flagged.

This rule has no parameters.

Example

```
package Foo is
   type Arr is array (1 .. 10) of Integer;
   type Acc is access Integer;

   A : array (1 .. 10) of Integer;  --  FLAG
   B : Arr;

   C : access Integer;              --  FLAG
   D : Acc;

   generic
      F1 : access Integer;          --  FLAG
      F2 : Acc;
   procedure Proc_G
     (P1 : access Integer;
      P2 : Acc);
end Foo;
```

`OTHERS_In_Aggregates`

Flag each use of an `others` choice in extension aggregates. In record and array aggregates, an `others` choice is flagged unless it is used to refer to all components, or to all but one component.

If, in case of a named array aggregate, there are two associations, one with an `others` choice and another with a discrete range, the `others` choice is flagged even if the discrete range specifies exactly one component; for example, `(1..1 => 0, others => 1)`.

This rule has no parameters.

Example

```
package Foo is
   type Arr is array (1 .. 10) of Integer;

   type Rec is record
      C1 : Integer;
      C2 : Integer;
      C3 : Integer;
      C4 : Integer;
   end record;

   type Tagged_Rec is tagged record
      C1 : Integer;
   end record;

   type New_Tagged_Rec is new Tagged_Rec with record
      C2 : Integer;
      C3 : Integer;
      C4 : Integer;
   end record;

   Arr_Var1 : Arr := (others => 1);
   Arr_Var2 : Arr := (1 => 1, 2=> 2, others => 0);  --  FLAG

   Rec_Var1 : Rec := (C1 => 1, others => 0);
   Rec_Var2 : Rec := (1, 2, others => 3);            --  FLAG

   Tagged_Rec_Var : Tagged_Rec := (C1 => 1);

   New_Tagged_Rec_Var : New_Tagged_Rec := (Tagged_Rec_Var with others => 0); -- FLAG
end Foo;
```

`OTHERS_In_CASE_Statements`

Flag any use of an `others` choice in a `case` statement.

This rule has no parameters.

Example

```
case J is
   when 1 =>
      I := I + 1;
   when 3 =>
      J := J + 1;
   when others =>        --  FLAG
      null;
end case;
```

`OTHERS_In_Exception_Handlers`

Flag any use of an `others` choice in an exception handler.

This rule has no parameters.

Example

```
exception
   when Constraint_Error =>
      I:= Integer'Last;
   when others =>                   --  FLAG
      I := I_Old;
      raise;
```

`Outbound_Protected_Assignments`

Flag an assignment statement located in a protected body if the variable name in the left part of the statement denotes an object declared outssided ourside this protected type or object.

This rule has no parameters.

Example

```
package Pack is
   Var : Integer;

   protected P is
      entry E (I : in out Integer);
      procedure P (I : Integer);
   private
      Flag : Boolean;
   end P;

end Pack;
package body Pack is
   protected body P is
      entry E (I : in out Integer) when Flag is
      begin
         I   := Var + I;
         Var := I;           --  FLAG
      end E;

      procedure P (I : Integer) is
      begin
         Flag := I > 0;
      end P;
   end P;
end Pack;
```

`Overly_Nested_Control_Structures`

Flag each control structure whose nesting level exceeds the value provided in the rule parameter.

The control structures checked are the following:

- `if` statement
- `case` statement
- `loop` statement
- selective accept statement
- timed entry call statement
- conditional entry call statement
- asynchronous select statement

The rule has the following parameter for the `+R` option:

N Positive integer specifying the maximal control structure nesting level that is not flagged

If the parameter for the `+R` option is not specified or if it is not a positive integer, `+R` option is ignored.

If more than one option is specified for the gnatcheck call, the later option and new parameter override the previous one(s).

Example

```
if I > 0 then
    for Idx in I .. J loop
       if J < 0 then
          case I is
             when 1 =>
                if Idx /= 0 then  --  FLAG (if rule parameter is 3)
                   J := J / Idx;
                end if;
             when others =>
                J := J + Idx;
          end case;
       end if;
    end loop;
end if;
```

`POS_On_Enumeration_Types`

Flag `'Pos` attribute in case if the attribute prefix has an enumeration type (including types derived from enumeration types).

This rule has no parameters.

Example

```
procedure Bar (Ch1, Ch2 : Character; I : in out Integer) is
begin
   if Ch1'Pos in 32 .. 126           --  FLAG
     and then
      Ch2'Pos not in 0 .. 31         --  FLAG
   then
      I := (Ch1'Pos + Ch2'Pos) / 2;  --  FLAG (twice)
   end if;
end Bar;
```

Positional_Actuals_For_Defaulted_Generic_Parameters

Flag each generic actual parameter corresponding to a generic formal parameter with a default initialization, if positional notation is used.

This rule has no parameters.

Example

```
package Foo is
   function Fun_1 (I : Integer) return Integer;
   function Fun_2 (I : Integer) return Integer;

   generic
      I_Par1 : Integer;
      I_Par2 : Integer := 1;
      with function Fun_1 (I : Integer) return Integer is <>;
      with function Fun_3 (I : Integer) return Integer is Fun_2;
   package Pack_G is
      Var_1 : Integer := I_Par1;
      Var_2 : Integer := I_Par2;
      Var_3 : Integer := Fun_1 (Var_1);
      Var_4 : Integer := Fun_3 (Var_2);
   end Pack_G;

   package Pack_I_1 is new Pack_G (1);

   package Pact_I_2 is new Pack_G
     (2, I_Par2 => 3, Fun_1 => Fun_2, Fun_3 => Fun_1);

   package Pack_I_3 is new Pack_G (1,
                                   2,             --  FLAG
                                   Fun_2,         --  FLAG
                                   Fun_1);        --  FLAG

end Foo;
```

Positional_Actuals_For_Defaulted_Parameters

Flag each actual parameter to a subprogram or entry call where the corresponding formal parameter has a default expression, if positional notation is used.

This rule has no parameters.

Example

```
   procedure Proc (I : in out Integer; J : Integer := 0) is
   begin
      I := I + J;
   end Proc;

begin
   Proc (Var1, Var2);    --  FLAG
```

`Positional_Components`

Flag each array, record and extension aggregate that includes positional notation.

This rule has no parameters.

Example

```
package Foo is
   type Arr is array (1 .. 10) of Integer;

   type Rec is record
      C_Int  : Integer;
      C_Bool : Boolean;
      C_Char : Character;
   end record;

   Var_Rec_1 : Rec := (C_Int => 1, C_Bool => True, C_Char => 'a');
   Var_Rec_2 : Rec := (2, C_Bool => False, C_Char => 'b');    --  FLAG
   Var_Rec_3 : Rec := (1, True, 'c');                         --  FLAG
end Foo;
```

`Positional_Generic_Parameters`

Flag each positional actual generic parameter except for the case when the generic unit being instantiated has exactly one generic formal parameter.

This rule has no parameters.

Example

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Conversion;
procedure Bar (I : in out Integer) is
   type My_Int is range -12345 .. 12345;

   function To_My_Int is new Ada.Unchecked_Conversion
     (Source => Integer, Target => My_Int);

   function To_Integer is new Ada.Unchecked_Conversion
```

```
     (My_Int, Integer);                                 --  FLAG (twice)

   package My_Int_IO is new  Ada.Text_IO.Integer_IO (My_Int);
```

Positional_Parameters

Flag each positional parameter notation in a subprogram or entry call, except for the following:

- Parameters of calls to attribute subprograms are not flagged;
- Parameters of prefix or infix calls to operator functions are not flagged;
- If the called subprogram or entry has only one formal parameter, the parameter of the call is not flagged;
- If a subprogram call uses the *Object.Operation* notation, then
  - the first parameter (that is, *Object*) is not flagged;
  - if the called subprogram has only two parameters, the second parameter of the call is not flagged;

This rule has the following (optional) parameters for the +R option:

All if this parameter is specified, all the positional parameter associations that can be replaced with named associations according to language rules are flagged

Example

```
procedure Bar (I : in out Integer) is
   function My_Max (Left, Right : Integer) return Integer renames Integer'Max;

   procedure Proc1 (I : in out Integer) is
   begin
      I := I + 1;
   end Proc1;

   procedure Proc2 (I, J : in out Integer) is
   begin
      I := I + J;
   end Proc2;

   L, M : Integer := 1;
begin
   Proc1 (L);
   Proc2 (L, M);                            --  FLAG (twice)
   Proc2 (I => M, J => L);

   L := Integer'Max (10, M);
   M := My_Max (100, Right => L);           --  FLAG

end Bar;
```

Recursive_Subprograms

Flags specs (and bodies that act as specs) of recursive subprograms. A subprogram is considered as recursive in a given context if there exists a chain of direct calls starting from the body of, and ending at this subprogram within this context. A context is provided by the set of Ada sources specified as arguments of a given gnatcheck call. Neither dispatching calls nor calls through access-to-subprograms are considered as direct calls by this rule.

Generic subprograms and subprograms detected in generic units are not flagged. Recursive subprograms in expanded generic instantiations are flagged.

This rule does not take into account subprogram calls in aspect definitions.

The rule has an optional parameters for `+R` option:

Skip_Dispatching_Calls Do not take into account dispatching calls when building and analyzing call chains.

Example

```
function Factorial (N : Natural) return Positive is  --  FLAG
begin
   if N = 0 then
      return 1;
   else
      return N * Factorial (N - 1);
   end if;
end Factorial;
```

`Single_Value_Enumeration_Types`

Flag an enumeration type definition if it contains a single enumeration literal specification

This rule has no parameters.

Example

```
type Enum3 is (A, B, C);
type Enum1 is (D);      --  FLAG
```

`Unchecked_Address_Conversions`

Flag instantiations of `Ada.Unchecked_Conversion` if the actual for the formal type Source is the `System.Address` type (or a type derived from it), and the actual for the formal type `Target` is an access type (including types derived from access types). This include cases when the actual for `Source` is a private type and its full declaration is a type derived from `System.Address`, and cases when the actual for `Target` is a private type and its full declaration is an access type. The rule is checked inside expanded generics.

This rule has no parameters.

Example

```
with Ada.Unchecked_Conversion;
with System;
package Foo is
   type My_Address is new System.Address;
   type My_Integer is new Integer;
   type My_Access is access all My_Integer;

   function Address_To_Access is new Ada.Unchecked_Conversion  --  FLAG
     (Source => My_Address,
      Target => My_Access);
end Foo;
```

Unchecked_Conversions_As_Actuals

Flag call to instantiation of `Ada.Unchecked_Conversion` if it is an actual in procedure or entry call or if it is a default value in a subprogram or entry parameter specification.

This rule has no parameters.

Example

```
with Ada.Unchecked_Conversion;
procedure Bar (I : in out Integer) is
   type T1 is array (1 .. 10) of Integer;
   type T2 is array (1 .. 10) of Integer;

   function UC is new Ada.Unchecked_Conversion (T1, T2);

   Var1 : T1 := (others => 1);
   Var2 : T2 := (others => 2);

   procedure Init (X : out T2; Y : T2 := UC (Var1)) is   --  FLAG
   begin
      X := Y;
   end Init;

   procedure Ident (X : T2; Y : out T2) is
   begin
      Y := X;
   end Ident;

begin
   Ident (UC (Var1), Var2);                              --  FLAG
end Bar;
```

Unconditional_Exits

Flag unconditional `exit` statements.

This rule has no parameters.

Example

```
procedure Find_A (S : String; Idx : out Natural) is
begin
   Idx := 0;

   for J in S'Range loop
      if S (J) = 'A' then
         Idx := J;
         exit;             --  FLAG
      end if;
   end loop;
end Find_A;
```

`Uninitialized_Global_Variables`

Flag an object declaration located immediately within a package declaration, a generic package declaration or a package body, if it does not have an explicit initialization. Do not flag deferred constant declarations and declarations of objects of limited types.

This rule has no parameters.

Example

```
package Foo is
   Var1 : Integer;       --  FLAG
   Var2 : Integer := 0;
end Foo;
```

`Unnamed_Blocks_And_Loops`

Flag each unnamed block statement. Flag a unnamed loop loop statement if this statement is enclosed by another loop statement or if it encloses another loop statement.

The rule has no parameters.

Example

```
procedure Bar (S : in out String) is
   I : Integer := 1;
begin
   if S'Length > 10 then
      declare                                    --  FLAG
         S1   : String (S'Range);
         Last : Positive := S1'Last;
         Idx  : Positive := 0;
      begin
         for J in S'Range loop                   --  FLAG
            S1 (Last - Idx) := S (J);
            Idx             := Idx + 1;
            for K in S'Range loop               --  FLAG
               S (K) := Character'Succ (S (K));
            end loop;

         end loop;

         S := S1;
      end;
   end if;
end Bar;
```

USE_PACKAGE_Clauses

Flag all `use` clauses for packages; `use type` clauses are not flagged.

This rule has no parameters.

Example

```
with Ada.Text_IO;
use Ada.Text_IO;                                --  FLAG
procedure Bar (S : in out String) is
```

9.1.6 Readability

The rules described in this subsection may be used to enforce feature usages that contribute towards readability.

Identifier_Casing

Flag each defining identifier that does not have a casing corresponding to the kind of entity being declared. All defining names are checked. For the defining names from the following kinds of declarations a special casing scheme can be defined:

- type and subtype declarations;
- enumeration literal specifications (not including character literals) and function renaming declarations if the renaming entity is an enumeration literal;
- constant and number declarations (including object renaming declarations if the renamed object is a constant);
- exception declarations and exception renaming declarations.

The rule may have the following parameters for `+R`:

- Type=*casing_scheme*

  Specifies casing for names from type and subtype declarations.

- Enum=*casing_scheme*

  Specifies the casing of defining enumeration literals and for the defining names in a function renaming declarations if the renamed entity is an enumeration literal.

- Constant=*casing_scheme*

  Specifies the casing for defining names from constants and named number declarations, including the object renaming declaration if the renamed object is a constant

- Exception=*casing_scheme*

  Specifies the casing for names from exception declarations and exception renaming declarations.

- Others=*casing_scheme*

  Specifies the casing for all defining names for which no special casing scheme is specified. If this parameter is not set, the casing for the entities that do not correspond to the specified parameters is not checked.

- Exclude=*dictionary_file*

  Specifies casing exceptions.

Where:

```
casing_scheme ::= upper|lower|mixed
```

upper means that the defining identifier should be upper-case. *lower* means that the defining identifier should be lower-case *mixed* means that the first defining identifier letter and the first letter after each underscore should be upper-case, and all the other letters should be lower-case

If a defining identifier is from a declaration for which a specific casing scheme can be set, but the corresponding parameter is not specified for the rule, then the casing scheme defined by `Others` parameter is used to check this identifier. If `Others` parameter also is not set, the identifier is not checked.

dictionary_file is the name of the text file that contains casing exceptions. The way how this rule is using the casing exception dictionary file is consistent with using the casing exception dictionary in the GNAT pretty-printer *gnatpp*, see GNAT User's Guide.

There are two kinds of exceptions:

identifier If a dictionary file contains an identifier, then each occurrence of that (defining) identifier in the checked source should use the casing specified included in *dictionary_file*

wildcard A wildcard has the following syntax

```
wildcard ::= *simple_identifier* |
                     *simple_identifier |
                     simple_identifier*
simple_identifier ::= letter{letter_or_digit}
```

`simple_identifier` specifies the casing of subwords (the term 'subword' is used below to denote the part of a name which is delimited by '_' or by the beginning or end of the word and which does not contain any '_' inside). A wildcard of the form `simple_identifier*` defines the casing of the first subword of a defining name to check, the wildcard of the form `*simple_identifier` specifies the casing of the last subword, and the wildcard of the form `*simple_identifier*` specifies the casing of any subword.

If for a defining identifier some of its subwords can be mapped onto wildcards, but some other cannot, the casing of the identifier subwords that are not mapped onto wildcards from casing exception dictionary is checked against the casing scheme defined for the corresponding entity.

If some identifier is included in the exception dictionary both as a whole identifier and can be mapped onto some wildcard from the dictionary, then it is the identifier and not the wildcard that is used to check the identifier casing.

If more than one dictionary file is specified, or a dictionary file contains more than one exception variant for the same identifier, the new casing exception overrides the previous one.

Casing check against dictionary file(s) has a higher priority than checks against the casing scheme specified for a given entity/declaration kind.

`+R` option should contain at least one parameter.

There is no parameter for `-R` option, it just turns the rule off.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are:

Type Exempts check for type and subtype name casing

Enum Exempts check for enumeration literal name casing

Constant Exempts check for constant name casing

Exception Exempts check for exception name casing

Others Exempts check for defining names for which no special casing scheme is specified.

Exclude Exempts check for defining names for which casing schemes are specified in exception dictionaries

Example

```
--  if the rule is activated as '+RIdentifier_Casing:Type=upper,others=mixed'
package Foo is
   type ENUM_1 is (A1, B1, C1);
   type Enum_2 is (A2, B2, C2);       --  FLAG

   Var1 : Enum_1 := A1;
   VAR2 : ENUM_2 := A2;               --  FLAG
end Foo;
```

`Identifier_Prefixes`

Flag each defining identifier that does not have a prefix corresponding to the kind of declaration it is defined by. The defining names in the following kinds of declarations are checked:

- type and subtype declarations (task, protected and access types are treated separately);
- enumeration literal specifications (not including character literals) and function renaming declarations if the renaming entity is an enumeration literal;
- exception declarations and exception renaming declarations;
- constant and number declarations (including object renaming declarations if the renamed object is a constant).

Defining names declared by single task declarations or single protected declarations are not checked by this rule.

The defining name from the full type declaration corresponding to a private type declaration or a private extension declaration is never flagged. A defining name from an incomplete type declaration is never flagged.

The defining name from a subprogram renaming-as-body declaration is never flagged.

For a deferred constant, the defining name in the corresponding full constant declaration is never flagged.

The defining name from a body that is a completion of a program unit declaration or a proper body of a subunit is never flagged.

The defining name from a body stub that is a completion of a program unit declaration is never flagged.

Note that the rule checks only defining names. Usage name occurrence are not checked and are never flagged.

The rule may have the following parameters:

- For the `+R` option:
- Type=*string*

  Specifies the prefix for a type or subtype name.

- Concurrent=*string*

  Specifies the prefix for a task and protected type/subtype name. If this parameter is set, it overrides for task and protected types the prefix set by the Type parameter.

- Access=*string*

  Specifies the prefix for an access type/subtype name. If this parameter is set, it overrides for access types the prefix set by the `Type` parameter.

- Class_Access=*string*

  Specifies the prefix for the name of an access type/subtype that points to some class-wide type. If this parameter is set, it overrides for such access types and subtypes the prefix set by the `Type` or `Access` parameter.

- Subprogram_Access=*string*

  Specifies the prefix for the name of an access type/subtype that points to a subprogram. If this parameter is set, it overrides for such access types/subtypes the prefix set by the `Type` or `Access` parameter.

- Derived=*string1:string2*

  Specifies the prefix for a type that is directly derived from a given type or from a subtype thereof. *string1* should be a full expanded Ada name of the ancestor type (starting from the full expanded compilation unit name), *string2* defines the prefix to check. If this parameter is set, it overrides for types that are directly derived from the given type the prefix set by the `Type` parameter.

- Constant=*string*

  Specifies the prefix for defining names from constants and named number declarations, including the object renaming declaration if the renamed object is a constant

- Enum=*string*

  Specifies the prefix for defining enumeration literals and for the defining names in a function renaming declarations if the renamed entity is an enumeration literal.

- Exception=*string*

  Specifies the prefix for defining names from exception declarations and exception renaming declarations.

Exclusive

  Check that only those kinds of names for which specific prefix is defined have that prefix (e.g., only type/subtype names have prefix *T_*, but not variable or package names), and flag all defining names that have any of the specified prefixes but do not belong to the kind of entities this prefix is defined for. By default the exclusive check mode is ON.

For the `-R` option:

All_Prefixes Removes all the prefixes specified for the identifier prefix checks, whether by default or as specified by other rule parameters and disables the rule.

Type Removes the prefix specified for type/subtype names. This does not remove prefixes specified for specific type kinds and does not disable checks for these specific kinds.

Concurrent Removes the prefix specified for task and protected types.

Access Removes the prefix specified for access types. This does not remove prefixes specified for specific access types (access to subprograms and class-wide access)

Class_Access Removes the prefix specified for access types pointing to class-wide types.

Subprogram_Access Removes the prefix specified for access types pointing to subprograms.

Derived Removes prefixes specified for derived types that are directly derived from specific types.

Constant Removes the prefix specified for constant and number names and turns off the check for these names.

Exception Removes the prefix specified for exception names and turns off the check for exception names.

Enum Removes the prefix specified for enumeration literal names and turns off the check for them.

Exclusive Turns of the check that only names of specific kinds of entities have prefixes specified for these kinds.

If more than one parameter is used, parameters must be separated by commas.

If more than one option is specified for the gnatcheck invocation, a new option overrides the previous one(s).

The `+RIdentifier_Prefixes` option (with no parameter) enables checks for all the name prefixes specified by previous options used for this rule. If no prefix is specified, the rule is not enabled.

The `-RIdentifier_Prefixes` option (with no parameter) disables all the checks but keeps all the prefixes specified by previous options used for this rule.

There is no default prefix setting for this rule. All checks for name prefixes are case-sensitive

If any error is detected in a rule parameter, that parameter is ignored. In such a case the options that are set for the rule are not specified.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are:

Type Exempts check for type and subtype name prefixes

Concurrent Exempts check for task and protected type/subtype name prefixes

Access Exempts check for access type/subtype name prefixes

Class_Access Exempts check for names of access types/subtypes that point to some class-wide types

Subprogram_Access Exempts check for names of access types/subtypes that point to subprograms

Derived Exempts check for derived type name prefixes

Constant Exempts check for constant and number name prefixes

Exception Exempts check for exception name prefixes

Enum Exempts check for enumeration literal name prefixes

Exclusive Exempts check that only names of specific kinds of entities have prefixes specified for these kinds

Example

```
--  if the rule is activated as '+RIdentifier_Prefixes:Type=Type_,Constant=Const_,ExceptioN=X_'
package Foo is
   type Type_Enum_1 is (A1, B1, C1);
   type Enum_2      is (A2, B2, C2);         --  FLAG

   Const_C1 : constant Type_Enum_1 := A1;
   Const2   : constant Enum_2      := A2;    --  FLAG
   X_Exc_1 : exception;
   Exc_2   : exception;                          --  FLAG
end Foo;
```

Identifier_Suffixes

Flag the declaration of each identifier that does not have a suffix corresponding to the kind of entity being declared. The following declarations are checked:

- type declarations
- subtype declarations
- object declarations (variable and constant declarations, but not number, declarations, record component declarations, parameter specifications, extended return object declarations, formal object declarations)
- package renaming declarations (but not generic package renaming declarations)

The default checks (enforced by the *Default* rule parameter) are:

- type-defining names end with `_T`, unless the type is an access type, in which case the suffix must be `_A`
- constant names end with `_C`
- names defining package renamings end with `_R`
- the check for access type objects is not enabled

Defining identifiers from incomplete type declarations are never flagged.

For a private type declaration (including private extensions), the defining identifier from the private type declaration is checked against the type suffix (even if the corresponding full declaration is an access type declaration), and the defining identifier from the corresponding full type declaration is not checked.

For a deferred constant, the defining name in the corresponding full constant declaration is not checked.

Defining names of formal types are not checked.

Check for the suffix of access type data objects is applied to the following kinds of declarations:

- variable and constant declaration
- record component declaration
- return object declaration
- parameter specification
- extended return object declaration
- formal object declaration

If both checks for constant suffixes and for access object suffixes are enabled, and if different suffixes are defined for them, then for constants of access type the check for access object suffixes is applied.

The rule may have the following parameters:

- For the `+R` option (unless the parameter is `Default`, then only the explicitly specified suffix is checked, and no defaults are used):

Default Sets the default listed above for all the names to be checked.

- Type_Suffix=*string*

Specifies the suffix for a type name.

- Access_Suffix=*string*

  Specifies the suffix for an access type name. If this parameter is set, it overrides for access types the suffix set by the `Type_Suffix` parameter. For access types, *string* may have the following format: *suffix1(suffix2)*. That means that an access type name should have the *suffix1* suffix except for the case when the designated type is also an access type, in this case the type name should have the *suffix1 & suffix2* suffix.

- Class_Access_Suffix=*string*

  Specifies the suffix for the name of an access type that points to some class-wide type. If this parameter is set, it overrides for such access types the suffix set by the `Type_Suffix` or `Access_Suffix` parameter.

- Class_Subtype_Suffix=*string*

  Specifies the suffix for the name of a subtype that denotes a class-wide type.

- Constant_Suffix=*string*

  Specifies the suffix for a constant name.

- Renaming_Suffix=*string*

  Specifies the suffix for a package renaming name.

- Access_Obj_Suffix=*string*

  Specifies the suffix for objects that have an access type (including types derived from access types).

- Interrupt_Suffix=*string*

  Specifies the suffix for protected subprograms used as interrupt handlers.

- For the `-R` option:

All_Suffixes Remove all the suffixes specified for the identifier suffix checks, whether by default or as specified by other rule parameters. All the checks for this rule are disabled as a result.

Type_Suffix Removes the suffix specified for types. This disables checks for types but does not disable any other checks for this rule (including the check for access type names if `Access_Suffix` is set).

Access_Suffix Removes the suffix specified for access types. This disables checks for access type names but does not disable any other checks for this rule. If `Type_Suffix` is set, access type names are checked as ordinary type names.

Class_Access_Suffix Removes the suffix specified for access types pointing to class-wide type. This disables specific checks for names of access types pointing to class-wide types but does not disable any other checks for this rule. If `Type_Suffix` is set, access type names are checked as ordinary type names. If `Access_Suffix` is set, these access types are checked as any other access type name.

Class_Subtype_Suffix Removes the suffix specified for subtype names. This disables checks for subtype names but does not disable any other checks for this rule.

Constant_Suffix Removes the suffix specified for constants. This disables checks for constant names but does not disable any other checks for this rule.

Renaming_Suffix Removes the suffix specified for package renamings. This disables checks for package renamings but does not disable any other checks for this rule.

Access_Obj_Suffix Removes the suffix specified for objects of access types, this disables checks for such objects. It does not disable any other checks for this rule

Interrupt_Suffix Removes the suffix specified for protected subprograms used as interrupt handlers. It does not disable any other checks for this rule.

If more than one parameter is used, parameters must be separated by commas.

If more than one option is specified for the *gnatcheck* invocation, a new option overrides the previous one(s).

The `+RIdentifier_Suffixes` option (with no parameter) enables checks for all the name suffixes specified by previous options used for this rule.

The `-RIdentifier_Suffixes` option (with no parameter) disables all the checks but keeps all the suffixes specified by previous options used for this rule.

The *string* value must be a valid suffix for an Ada identifier (after trimming all the leading and trailing space characters, if any). Parameters are not case sensitive, except the *string* part.

If any error is detected in a rule parameter, the parameter is ignored. In such a case the options that are set for the rule are not specified.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are:

Type Exempts check for type name suffixes

Access Exempts check for access type name suffixes

Access_Obj Exempts check for access object name suffixes

Class_Access Exempts check for names of access types that point to some class-wide types

Class_Subtype Exempts check for names of subtypes that denote class-wide types

Constant Exempts check for constant name suffixes

Renaming Exempts check for package renaming name suffixes

Example

```
--  if the rule is activated as '+RIdentifier_Suffixes:Access_Suffix=_PTR,Type_Suffix=_T,Constant_Su
package Foo is
   type Int   is range 0 .. 100;       --  FLAG
   type Int_T is range 0 .. 100;

   type Int_A   is access Int;         --  FLAG
   type Int_PTR is access Int;

   Const   : constant Int := 1;        --  FLAG
   Const_C : constant Int := 1;

end Foo;
```

`Max_Identifier_Length`

Flag any defining identifier that has length longer than specified by the rule parameter. The rule has a mandatory parameter for +R option:

N The maximal allowed identifier length specification.

Example

```
type My_Type is range -100 .. 100;
My_Variable_With_A_Long_Name : My_Type;  -- FLAG (if rule parameter is 27 or smaller)
```

Misnamed_Controlling_Parameters

Flag a declaration of a dispatching operation, if the first parameter is not a controlling one and its name is not `This` (the check for parameter name is not case-sensitive). Declarations of dispatching functions with a controlling result and no controlling parameter are never flagged.

A subprogram body declaration, subprogram renaming declaration, or subprogram body stub is flagged only if it is not a completion of a prior subprogram declaration.

This rule has no parameters.

Example

```
package Foo is
   type T is tagged private;

   procedure P1 (This : in out T);
   procedure P2 (That : in out T);               --  FLAG
   procedure P1 (I : Integer; This : in out T); --  FLAG
```

Name_Clashes

Check that certain names are not used as defining identifiers. The names that should not be used as identifiers must be listed in a dictionary file that is a rule parameter. A defining identifier is flagged if it is included in a dictionary file specified as a rule parameter, the check is not case-sensitive. More than one dictionary file can be specified as the rule parameter, in this case the rule checks defining identifiers against the union of all the identifiers from all the dictionary files provided as the rule parameters.

This rule has the following (mandatory) parameters for the `+R` option:

dictionary_file The name of a dictionary file.

A dictionary file is a plain text file. The maximum line length for this file is 1024 characters. If the line is longer than this limit, extra characters are ignored.

If the name of the dictionary file does not contain any path information and the rule option is specifies in a rule file, first the tool tries to locate the dictionary file in the same directory where the rule file is located, and if the attempt fails - in the current directory.

Each line can be either an empty line, a comment line, or a line containing a list of identifiers separated by space or HT characters. A comment is an Ada-style comment (from `--` to end-of-line). Identifiers must follow the Ada syntax for identifiers. A line containing one or more identifiers may end with a comment.

Example

```
--  If the dictionary file contains names 'One' and 'Two":
One          : constant Integer := 1;      --  FLAG
Two          : constant Float   := 2.0;    --  FLAG
Constant_One : constant Float   := 1.0;
```

Object_Declarations_Out_Of_Order

Flag any object declaration that is located in a library unit body if this is preceding by a declaration of a program unit spec, stub or body.

This rule has no parameters.

Example

```
procedure Proc is
   procedure Proc1 is separate;

   I : Integer;    -- FLAG
```

One_Construct_Per_Line

Flag any statement, declaration or representation clause if the code line where this construct starts contains some other Ada code symbols preceding or following this construct. The following constructs are not flagged:

- enumeration literal specification;
- parameter specifications;
- discriminant specifications;
- mod clauses;
- loop parameter specification;
- entry index specification;
- choice parameter specification;

In case if we have two or more declarations/statements/clauses on a line and if there is no Ada code preceding the first construct, the first construct is flagged

This rule has no parameters.

Example

```
procedure Swap (I, J : in out Integer) is
   Tmp : Integer;
begin
   Tmp := I;
   I := J; J := Tmp;      --  FLAG
end Swap;
```

Uncommented_BEGIN_In_Package_Bodies

Flags each package body with declarations and a statement part that does not include a trailing comment on the line containing the `begin` keyword; this trailing comment needs to specify the package name and nothing else. The `begin` is not flagged if the package body does not contain any declarations.

If the `begin` keyword is placed on the same line as the last declaration or the first statement, it is flagged independently of whether the line contains a trailing comment. The diagnostic message is attached to the line containing the first statement.

This rule has no parameters.

Example

```
package body Foo is
   procedure Proc (I : out Integer) is
   begin
      I := Var;
   end Proc;

   package body Inner is
      procedure Inner_Proc (I : out Integer) is
      begin
         I := Inner_Var;
      end  ;
   begin  -- Inner
      Inner_Var := 1;
   end Inner;
begin                 --  FLAG
   Var := Inner.Inner_Var + 1;
end Foo;
```

9.1.7 Source Code Presentation

This subsection is a placeholder; there are currently no rules in this category.

9.2 Feature Usage Rules

The rules in this section can be used to enforce specific usage patterns for a variety of language features.

9.2.1 Abort_Statements

Flag abort statements.

This rule has no parameters.

Example

```
if Flag then
   abort T;    --  FLAG
end if;
```

9.2.2 `Abstract_Type_Declarations`

Flag all declarations of abstract types. For an abstract private type, both the private and full type declarations are flagged.

This rule has no parameters.

Example

```
package Foo is
   type Figure is abstract tagged private;              --  FLAG
   procedure Move (X : in out Figure) is abstract;
private
   type Figure is abstract tagged null record;          --  FLAG
end Foo;
```

9.2.3 `Anonymous_Subtypes`

Flag all uses of anonymous subtypes except for the following:

- when the subtype indication depends on a discriminant, this includes the cases of a record component definitions when a component depends on a discriminant, and using the discriminant of the derived type to constraint the parent type;
- when a self-referenced data structure is defined, and a discriminant is constrained by the reference to the current instance of a type;

A use of an anonymous subtype is any instance of a subtype indication with a constraint, other than one that occurs immediately within a subtype declaration. Any use of a range other than as a constraint used immediately within a subtype declaration is considered as an anonymous subtype.

The rule does not flag ranges in the component clauses from a record representation clause, because the language rules do not allow to use subtype names there.

An effect of this rule is that `for` loops such as the following are flagged (since `1..N` is formally a 'range'):

Declaring an explicit subtype solves the problem:

This rule has no parameters.

9.2.4 `Blocks`

Flag each block statement.

This rule has no parameters.

Example

```
if I /= J then
   declare             --  FLAG
      Tmp : Integer;
   begin
      TMP := I;
      I   := J;
      J   := Tmp;
   end;
end if;
```

9.2.5 Complex_Inlined_Subprograms

Flag a subprogram (or generic subprogram, or instantiation of a subprogram) if pragma Inline is applied to it and at least one of the following conditions is met:

- it contains at least one complex declaration such as a subprogram body, package, task, protected declaration, or a generic instantiation (except instantiation of `Ada.Unchecked_Conversion`);
- it contains at least one complex statement such as a loop, a case or an if statement;
- the number of statements exceeds a value specified by the *N* rule parameter;

Subprogram renamings are also considered.

This rule has the following (mandatory) parameter for the `+R` option:

N Positive integer specifying the maximum allowed total number of statements in the subprogram body.

Example

```
procedure Swap (I, J : in out Integer) with Inline => True;

procedure Swap (I, J : in out Integer) is   --  FLAG
begin

   if I /= J then
      declare
         Tmp : Integer;
      begin
         TMP := I;
         I   := J;
         J   := Tmp;
      end;
   end if;

end Swap;
```

9.2.6 Conditional_Expressions

Flag use of conditional expression.

This rule has the following (optional) parameters for the `+R` option:

Except_Assertions Do not flag a conditional expression if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- `Assert`

GNAT-specific

- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Invariant`
- `Loop_Invariant`
- `Loop_Variant`
- `Postcondition`
- `Precondition`
- `Predicate`
- `Refined_Post`

definition of the following aspects

Language-defined

- `Static_Predicate`
- `Dynamic_Predicate`
- `Pre`
- `Pre'Class`
- `Post`
- `Post'Class`
- `Type_Invariant`
- `Type_Invariant'Class`

GNAT-specific

- `Contract_Cases`
- `Invariant`
- `Invariant'Class`
- `Predicate`
- `Refined_Post`

Example

```
Var1 : Integer := (if I > J then 1 else 0);  --  FLAG
Var2 : Integer := I + J;
```

9.2.7 `Controlled_Type_Declarations`

Flag all declarations of controlled types. A declaration of a private type is flagged if its full declaration declares a controlled type. A declaration of a derived type is flagged if its ancestor type is controlled. Subtype declarations are not checked. A declaration of a type that itself is not a descendant of a type declared in `Ada.Finalization` but has a controlled component is not checked.

This rule has no parameters.

Example

```
with Ada.Finalization;
package Foo is
   type Resource is new Ada.Finalization.Controlled with private;  --  FLAG
```

9.2.8 `Declarations_In_Blocks`

Flag all block statements containing local declarations. A `declare` block with an empty *declarative_part* or with a *declarative part* containing only pragmas and/or `use` clauses is not flagged.

This rule has no parameters.

Example

```
if I /= J then
   declare                       --  FLAG
      Tmp : Integer;
   begin
      TMP := I;
      I   := J;
      J   := Tmp;
   end;
end if;
```

9.2.9 `Deeply_Nested_Inlining`

Flag a subprogram (or generic subprogram) if pragma Inline has been applied to it, and it calls another subprogram to which pragma Inline applies, resulting in potential nested inlining, with a nesting depth exceeding the value specified by the *N* rule parameter.

This rule requires the global analysis of all the compilation units that are *gnatcheck* arguments; such analysis may affect the tool's performance.

This rule has the following (mandatory) parameter for the `+R` option:

N Positive integer specifying the maximum level of nested calls to subprograms to which pragma Inline has been applied.

Example

```
procedure P1 (I : in out integer) with Inline => True;    --  FLAG
procedure P2 (I : in out integer) with Inline => True;
procedure P3 (I : in out integer) with Inline => True;
procedure P4 (I : in out integer) with Inline => True;

procedure P1 (I : in out integer) is
begin
   I := I + 1;
   P2 (I);
end;

procedure P2 (I : in out integer) is
begin
   I := I + 1;
   P3 (I);
end;

procedure P3 (I : in out integer) is
begin
   I := I + 1;
   P4 (I);
end;

procedure P4 (I : in out integer) is
begin
   I := I + 1;
end;
```

9.2.10 `Default_Parameters`

Flag all default expressions in parameters specifications. All parameter specifications are checked: in subprograms (including formal, generic and protected subprograms) and in task and protected entries (including accept statements and entry bodies).

This rule has no parameters.

Example

```
procedure P (I : in out Integer; J : Integer := 0);    --  FLAG
procedure Q (I : in out Integer; J : Integer);
```

9.2.11 `Discriminated_Records`

Flag all declarations of record types with discriminants. Only the declarations of record and record extension types are checked. Incomplete, formal, private, derived and private extension type declarations are not checked. Task and protected type declarations also are not checked.

This rule has no parameters.

Example

```
type Idx is range 1 .. 100;
type Arr is array (Idx range <>) of Integer;
subtype Arr_10 is Arr (1 .. 10);

type Rec_1 (D : Idx) is record        --  FLAG
   A : Arr (1 .. D);
end record;

type Rec_2 (D : Idx) is record        --  FLAG
   B : Boolean;
end record;

type Rec_3 is record
   B : Boolean;
end record;
```

9.2.12 `Explicit_Full_Discrete_Ranges`

Flag each discrete range that has the form `A'First .. A'Last`.

This rule has no parameters.

Example

```
   subtype Idx is Integer range 1 .. 100;
begin
   for J in Idx'First .. Idx'Last loop   --  FLAG
      K := K + J;
   end loop;

   for J in Idx loop
      L := L + J;
   end loop;
```

9.2.13 `Expression_Functions`

Flag each expression function declared in a package specification (including specification of local packages and generic package specifications).

This rule has no parameters.

Example

```
package Foo is

   function F (I : Integer) return Integer is   --  FLAG
     (if I > 0 then I - 1 else I + 1);
```

9.2.14 `Fixed_Equality_Checks`

Flag all calls to the predefined equality operations for fixed-point types. Both '=' and '/=' operations are checked. User-defined equality operations are not flagged, nor are uses of operators that are renamings of the predefined equality operations. Also, the '=' and '/=' operations for floating-point types are not flagged.

This rule has no parameters.

Example

```
package Pack is
     type Speed is delta 0.01 range 0.0 .. 10_000.0;
     function Get_Speed return Speed;
end Pack;

with Pack; use Pack;
procedure Process is
     Speed1 : Speed := Get_Speed;
     Speed2 : Speed := Get_Speed;

     Flag : Boolean := Speed1 = Speed2;      --  FLAG
```

9.2.15 `Float_Equality_Checks`

Flag all calls to the predefined equality operations for floating-point types and private types whose completions are floating-point types. Both '=' and '/=' operations are checked. User-defined equality operations are not flagged, nor are uses of operators that are renamings of the predefined equality operations. Also, the '=' and '/=' operations for fixed-point types are not flagged.

This rule has no parameters.

Example

```
package Pack is
     type Speed is digits 0.01 range 0.0 .. 10_000.0;
     function Get_Speed return Speed;
end Pack;

with Pack; use Pack;
procedure Process is
     Speed1 : Speed := Get_Speed;
     Speed2 : Speed := Get_Speed;

     Flag : Boolean := Speed1 = Speed2;      --  FLAG
```

9.2.16 `Function_Style_Procedures`

Flag each procedure that can be rewritten as a function. A procedure can be converted into a function if it has exactly one parameter of mode `out` and no parameters of mode `in out`. Procedure declarations, formal procedure declarations, and generic procedure declarations are always checked. Procedure bodies and body stubs are flagged only if they do not have corresponding separate declarations. Procedure renamings and procedure instantiations are not flagged.

If a procedure can be rewritten as a function, but its `out` parameter is of a limited type, it is not flagged.

Protected procedures are not flagged. Null procedures also are not flagged.

This rule has no parameters.

Example

```
procedure Cannot_be_a_function (A, B : out Boolean);
procedure Can_be_a_function (A : out Boolean);           --  FLAG
```

9.2.17 `Generics_In_Subprograms`

Flag each declaration of a generic unit in a subprogram. Generic declarations in the bodies of generic subprograms are also flagged. A generic unit nested in another generic unit is not flagged. If a generic unit is declared in a local package that is declared in a subprogram body, the generic unit is flagged.

This rule has no parameters.

Example

```
procedure Proc is

   generic                                    --  FLAG
      type FT is range <>;
   function F_G (I : FT) return FT;
```

9.2.18 `Implicit_IN_Mode_Parameters`

Flag each occurrence of a formal parameter with an implicit `in` mode. Note that `access` parameters, although they technically behave like `in` parameters, are not flagged.

This rule has no parameters.

Example

```
procedure Proc1 (I :    Integer);           --  FLAG
procedure Proc2 (I : in Integer);
procedure Proc3 (I :    access Integer);
```

9.2.19 `Improperly_Located_Instantiations`

Flag all generic instantiations in library-level package specs (including library generic packages) and in all subprogram bodies.

Instantiations in task and entry bodies are not flagged. Instantiations in the bodies of protected subprograms are flagged.

This rule has no parameters.

Example

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Proc is
   package My_Int_IO is new Integer_IO (Integer);    --  FLAG
```

9.2.20 `Library_Level_Subprograms`

Flag all library-level subprograms (including generic subprogram instantiations).

This rule has no parameters.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Proc is                                --  FLAG
```

9.2.21 `Membership_Tests`

Flag use of membership test expression.

This rule has the following (optional) parameters for the `+R` option:

Multi_Alternative_Only Flag only those membership test expressions that have more than one membership choice in the membership choice list.

Float_Types_Only Flag only those membership test expressions that checks objects of floating point type and private types whose completions are floating-point types.

Except_Assertions Do not flag a membership test expression if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- `Assert`

GNAT-specific

- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Invariant`
- `Loop_Invariant`

- `Loop_Variant`
- `Postcondition`
- `Precondition`
- `Predicate`
- `Refined_Post`

definition of the following aspects

Language-defined

- `Static_Predicate`
- `Dynamic_Predicate`
- `Pre`
- `Pre'Class`
- `Post`
- `Post'Class`
- `Type_Invariant`
- `Type_Invariant'Class`

GNAT-specific

- `Contract_Cases`
- `Invariant`
- `Invariant'Class`
- `Predicate`
- `Refined_Post`

These three parameters are independent on each other.

Example

```
procedure Proc (S : in out Speed) is
begin
   if S in Low .. High then      --  FLAG
```

9.2.22 `Non_Qualified_Aggregates`

Flag each non-qualified aggregate. A non-qualified aggregate is an aggregate that is not the expression of a qualified expression. A string literal is not considered an aggregate, but an array aggregate of a string type is considered as a normal aggregate. Aggregates of anonymous array types are not flagged.

This rule has no parameters.

Example

```
type Arr is array (1 .. 10) of Integer;

Var1 : Arr := (1 => 10, 2 => 20, others => 30);             --  FLAG
Var2 : array (1 .. 10) of Integer := (1 => 10, 2 => 20, others => 30);
```

9.2.23 `Number_Declarations`

Number declarations are flagged.

This rule has no parameters.

Example

```
Num1 : constant := 13;                   --  FLAG
Num2 : constant := 1.3;                  --  FLAG

Const1 : constant Integer := 13;
Const2 : constant Float := 1.3;
```

9.2.24 `Numeric_Indexing`

Flag numeric literals, including those preceded by a predefined unary minus, if they are used as index expressions in array components. Literals that are subcomponents of index expressions are not flagged (other than the aforementioned case of unary minus).

This rule has no parameters.

Example

```
procedure Proc is
   type Arr is array (1 .. 10) of Integer;
   Var : Arr;
begin
   Var (1) := 10;       --  FLAG
```

9.2.25 `Numeric_Literals`

Flag each use of a numeric literal except for the following:

- a literal occurring in the initialization expression for a constant declaration or a named number declaration, or
- a literal occurring in an aspect definition or in an aspect clause, or
- an integer literal that is less than or equal to a value specified by the *N* rule parameter.
- a literal occurring in a declaration in case the *Statements_Only* rule parameter is given

This rule may have the following parameters for the `+R` option:

N *N* is an integer literal used as the maximal value that is not flagged (i.e., integer literals not exceeding this value are allowed)

`ALL` All integer literals are flagged

`Statements_Only` Numeric literals are flagged only when used in statements

If no parameters are set, the maximum unflagged value is 1, and the check for literals is not limited by statements only.

The last specified check limit (or the fact that there is no limit at all) is used when multiple `+R` options appear.

The `-R` option for this rule has no parameters. It disables the rule and restores its default operation mode. If the `+R` option subsequently appears, will be 1, and the check will not be limited by statements only.

Example

```
C1 : constant Integer := 10;
V1 :          Integer := C1;
V2 :          Integer := 20;      --  FLAG
```

9.2.26 `Parameters_Out_Of_Order`

Flag each subprogram and entry declaration whose formal parameters are not ordered according to the following scheme:

- `in` and `access` parameters first, then `in out` parameters, and then `out` parameters;
- for `in` mode, parameters with default initialization expressions occur last

Only the first violation of the described order is flagged.

The following constructs are checked:

- subprogram declarations (including null procedures);
- generic subprogram declarations;
- formal subprogram declarations;
- entry declarations;
- subprogram bodies and subprogram body stubs that do not have separate specifications

Subprogram renamings are not checked.

This rule has no parameters.

Example

```
procedure Proc1 (I : in out Integer; B : Boolean) is    --  FLAG
```

9.2.27 `Predicate_Testing`

Flag a subtype mark if it denotes a subtype defined with (static or dynamic) subtype predicate and is used as a membership choice in a membership test expression.

Flags ‘Valid attribute reference if the nominal subtype of the attribute prefix has (static or dynamic) subtype predicate.

This rule has the following (optional) parameters for the `+R` option:

Except_Assertions Do not flag a construct described above if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- `Assert`

GNAT-specific

- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Invariant`
- `Loop_Invariant`
- `Loop_Variant`
- `Postcondition`
- `Precondition`
- `Predicate`
- `Refined_Post`

definition of the following aspects

Language-defined

- `Static_Predicate`
- `Dynamic_Predicate`
- `Pre`
- `Pre'Class`
- `Post`
- `Post'Class`
- `Type_Invariant`
- `Type_Invariant'Class`

GNAT-specific

- `Contract_Cases`
- `Invariant`
- `Invariant'Class`
- `Predicate`
- `Refined_Post`

Example

```
with Support; use Support;
package Pack is
   subtype Even is Integer with Dynamic_Predicate => Even mod 2 = 0;

   subtype Small_Even is Even range -100 .. 100;

   B1 : Boolean := Ident (101) in Small_Even;       --  FLAG
```

9.2.28 `Relative_Delay_Statements`

Relative delay statements are flagged. Delay until statements are not flagged.

This rule has no parameters.

Example

```
if I > 0 then
   delay until Current_Time + Big_Delay;
else
   delay Small_Delay;                            --  FLAG
end if;
```

9.2.29 `Representation_Specifications`

Flag each record representation clause, enumeration representation clause and representation attribute clause. Flag each aspect definition that defines a representation aspect. Also flag any pragma that is classified by the Ada Standard as a representation pragma, and the definition of the corresponding aspects.

This rule has no parameters.

Example

```
type State         is (A,M,W,P);
type Mode          is (Fix, Dec, Exp, Signif);

type Byte_Mask     is array (0..7)  of Boolean
  with Component_Size => 1;                                 --  FLAG

type State_Mask    is array (State) of Boolean
  with Component_Size => 1;                                 --  FLAG

type Mode_Mask     is array (Mode)  of Boolean;
for Mode_Mask'Component_Size use 1;                         --  FLAG
```

9.2.30 `Quantified_Expressions`

Flag use of quantified expression.

This rule has the following (optional) parameters for the `+R` option:

Except_Assertions Do not flag a conditional expression if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- `Assert`

GNAT-specific

- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Invariant`
- `Loop_Invariant`
- `Loop_Variant`
- `Postcondition`
- `Precondition`
- `Predicate`
- `Refined_Post`

definition of the following aspects

Language-defined

- `Static_Predicate`
- `Dynamic_Predicate`
- `Pre`
- `Pre'Class`
- `Post`
- `Post'Class`
- `Type_Invariant`
- `Type_Invariant'Class`

GNAT-specific

- `Contract_Cases`
- `Invariant`
- `Invariant'Class`
- `Predicate`
- `Refined_Post`

Example

```
subtype Ind is Integer range 1 .. 10;
type Matrix is array (Ind, Ind) of Integer;

function Check_Matrix (M : Matrix) return Boolean is
  (for some I in Ind =>                              --  FLAG
     (for all J in Ind => M (I, J) = 0));            --  FLAG
```

9.2.31 `Raising_Predefined_Exceptions`

Flag each `raise` statement that raises a predefined exception (i.e., one of the exceptions `Constraint_Error`, `Numeric_Error`, `Program_Error`, `Storage_Error`, or `Tasking_Error`).

This rule has no parameters.

Example

```
begin
   raise Constraint_Error;     --  FLAG
```

9.2.32 `Subprogram_Access`

Flag all constructs that belong to access_to_subprogram_definition syntax category, and all access definitions that define access to subprogram.

This rule has no parameters.

Example

```
type Proc_A is access procedure ( I : Integer);       --  FLAG

procedure Proc
  (I       : Integer;
   Process : access procedure (J : in out Integer));   --  FLAG
```

9.2.33 `Too_Many_Dependencies`

Flag a library item or a subunit that immediately depends on more than N library units (N is a rule parameter). In case of a dependency on child units, implicit or explicit dependencies on all their parents are not counted.

This rule has the following (mandatory) parameters for the `+R` option:

N Positive integer specifying the maximal number of dependencies when the library item or subunit is not flagged.

Example

```
--  if rule parameter is 5 or smaller:
with Pack1;
with Pack2;
with Pack3;
with Pack4;
with Pack5;
with Pack6;
procedure Main is                  --  FLAG
```

9.2.34 `Unassigned_OUT_Parameters`

Flag procedures' `out` parameters that are not assigned.

An `out` parameter is flagged if the *sequence of statements* of the procedure body (before the procedure body's exception part, if any) contains no assignment to the parameter.

An `out` parameter is flagged in an *exception handler* in the exception part of the procedure body, if the *exception handler* contains neither an assignment to the parameter nor a raise statement.

Bodies of generic procedures are also considered.

The following are treated as assignments to an `out` parameter:

- an assignment statement, with the parameter or some component as the target
- passing the parameter (or one of its components) as an `out` or `in out` parameter, except for the case when it is passed to the call of an attribute subprogram.

This rule has no parameters.

Warning: This rule only detects a trivial case of an unassigned variable and doesn't provide a guarantee that there is no uninitialized access. The rule does not check function parameters (starting from Ada 2012 functions can have `out` parameters). It is not a replacement for rigorous check for uninitialized access provided by advanced static analysis tools.

Example

```
procedure Proc                   --  FLAG
   (I    : Integer;
    Out1 : out Integer;
    Out2 : out Integer)
is
begin
    Out1 := I + 1;
end Proc;
```

9.2.35 `Unconstrained_Array_Returns`

Flag each function returning an unconstrained array. Function declarations, function bodies (and body stubs) having no separate specifications, and generic function instantiations are flagged. Function calls and function renamings are not flagged.

Generic function declarations, and function declarations in generic packages, are not flagged. Instead, this rule flags the results of generic instantiations (that is, expanded specification and expanded body corresponding to an instantiation).

This rule has the following (optional) parameters for the `+R` option:

Except_String Do not flag functions that return the predefined `String` type or a type derived from it, directly or indirectly.

Example

```
type Arr is array (Integer range <>) of Integer;
subtype Arr_S is Arr (1 .. 10);

function F1 (I : Integer) return Arr;      --  FLAG
function F2 (I : Integer) return Arr_S;
```

9.2.36 `Unconstrained_Arrays`

Unconstrained array definitions are flagged.

This rule has no parameters.

Example

```
type Idx is range -100 .. 100;

type U_Arr is array (Idx range <>) of Integer;      --  FLAG
type C_Arr is array (Idx) of Integer;
```

9.3 Metrics-Related Rules

The rules in this section can be used to enforce compliance with specific code metrics, by checking that the metrics computed for a program lie within user-specifiable bounds. Depending on the metric, there may be a lower bound, an upper bound, or both. A construct is flagged if the value of the metric exceeds the upper bound or is less than the lower bound.

The name of any metrics rule consists of the prefix `Metrics_` followed by the name of the corresponding metric: `Essential_Complexity`, `Cyclomatic_Complexity`, or `LSLOC`. (The ‘LSLOC’ acronym stands for ‘Logical Source Lines Of Code’.) The meaning and the computed values of the metrics are the same as in *gnatmetric*.

For the `+R` option, each metrics rule has a numeric parameter specifying the bound (integer or real, depending on a metric). The `-R` option for the metrics rules does not have a parameter.

Example: the rule

```
+RMetrics_Cyclomatic_Complexity : 7
```

means that all bodies with cyclomatic complexity exceeding 7 will be flagged.

To turn OFF the check for cyclomatic complexity metric, use the following option:

```
-RMetrics_Cyclomatic_Complexity
```

9.3.1 `Metrics_Essential_Complexity`

The `Metrics_Essential_Complexity` rule takes a positive integer as upper bound. A program unit that is an executable body exceeding this limit will be flagged.

The Ada essential complexity metric is a McCabe cyclomatic complexity metric counted for the code that is reduced by excluding all the pure structural Ada control statements.

Example

```
--  if the rule parameter is 3 or less
procedure Proc (I : in out Integer; S : String) is   --  FLAG
begin
   if I in 1 .. 10 then
      for J in S'Range loop

         if S (J) = ' ' then
            if I > 10 then
               exit;
            else
               I := 10;
            end if;
         end if;

         I := I + Character'Pos (S (J));
      end loop;
   end if;
end Proc;
```

9.3.2 `Metrics_Cyclomatic_Complexity`

The `Metrics_Cyclomatic_Complexity` rule takes a positive integer as upper bound. A program unit that is an executable body exceeding this limit will be flagged.

The McCabe cyclomatic complexity metric is defined in http://www.mccabe.com/pdf/mccabe-nist235r.pdf The goal of cyclomatic complexity metric is to estimate the number of independent paths in the control flow graph that in turn gives the number of tests needed to satisfy paths coverage testing completeness criterion.

Example

```
--  if the rule parameter is 6 or less
procedure Proc (I : in out Integer; S : String) is   --  FLAG
begin
   if I in 1 .. 10 then
      for J in S'Range loop

         if S (J) = ' ' then
            if I < 10 then
```

```
               I := 10;
            end if;
         end if;

         I := I + Character'Pos (S (J));
      end loop;
   elsif S = "abs" then
      if I > 0 then
         I := I + 1;
      end if;
   end if;
end Proc;
```

9.3.3 `Metrics_LSLOC`

The `Metrics_LSLOC` rule takes a positive integer as upper bound. A program unit declaration or a program unit body exceeding this limit will be flagged.

The metric counts the total number of declarations and the total number of statements.

This rule contains optional parameters for `+R` option that allows to restrict the rule to specific constructs:

Subprograms Check the rule for subprogram bodies only.

Example

```
--  if the rule parameter is 20 or less
package Pack is                                 --  FLAG
   procedure Proc1 (I : in out Integer);
   procedure Proc2 (I : in out Integer);
   procedure Proc3 (I : in out Integer);
   procedure Proc4 (I : in out Integer);
   procedure Proc5 (I : in out Integer);
   procedure Proc6 (I : in out Integer);
   procedure Proc7 (I : in out Integer);
   procedure Proc8 (I : in out Integer);
   procedure Proc9 (I : in out Integer);
   procedure Proc10 (I : in out Integer);
end Pack;
```

9.4 SPARK Ada Rules

The rules in this section can be used to enforce compliance with the Ada subset allowed by the SPARK tools.

9.4.1 `Annotated_Comments`

Flags comments that are used as annotations or as special sentinels/markers. Such comments have the following structure

```
--<special_character> <comment_marker>
```

where

<special_character> character (such as '#', '$', '|' etc.) indicating that the comment is used for a specific purpose

<comment_marker> a word identifying the annotation or special usage (word here is any sequence of characters except white space)

There may be any amount of white space (including none at all) between `<special_character>` and `<comment_marker>`, but no white space is permitted between `'--'` and `<special_character>`. (A white space here is either a space character or horizontal tabulation)

`<comment_marker>` must not contain any white space.

`<comment_marker>` may be empty, in which case the rule flags each comment that starts with `--<special_character>` and that does not contain any other character except white space

The rule has the following (mandatory) parameter for the `+R` option:

S String with the following interpretation: the first character is the special comment character, and the rest is the comment marker. S must not contain white space.

The `-R` option erases all definitions of special comment annotations specified by the previous +R options.

The rule is case-sensitive.

Example:

The rule

```
+RAnnotated_Comments:#hide
```

will flag the following comment lines

```
--#hide
--# hide
--#           hide

   I := I + 1; --# hide
```

But the line

```
-- # hide
```

will not be flagged, because of the space between '–' and '#'.

The line

```
--#Hide
```

will not be flagged, because the string parameter is case sensitive.

9.4.2 `Boolean_Relational_Operators`

Flag each call to a predefined relational operator ('<', '>', '<=', '>=', '=' and '/=') for the predefined Boolean type. (This rule is useful in enforcing the SPARK language restrictions.)

Calls to predefined relational operators of any type derived from `Standard.Boolean` are not detected. Calls to user-defined functions with these designators, and uses of operators that are renamings of the predefined relational operators for `Standard.Boolean`, are likewise not detected.

This rule has no parameters.

Example

```
   procedure Proc (Flag_1 : Boolean; Flag_2 : Boolean; I : in out Integer) is
   begin
      if Flag_1 >= Flag_2 then      --  FLAG
```

9.4.3 `Expanded_Loop_Exit_Names`

Flag all expanded loop names in `exit` statements.

This rule has no parameters.

Example

```
procedure Proc (S : in out String) is
begin
   Search : for J in S'Range loop
      if S (J) = ' ' then
         S (J) := '_';
         exit Proc.Search;              --  FLAG
      end if;
   end loop Search;
end Proc;
```

9.4.4 `Non_SPARK_Attributes`

The SPARK language defines the following subset of Ada 95 attribute designators as those that can be used in SPARK programs. The use of any other attribute is flagged.

- `'Adjacent`
- `'Aft`
- `'Base`
- `'Ceiling`
- `'Component_Size`
- `'Compose`
- `'Copy_Sign`
- `'Delta`
- `'Denorm`
- `'Digits`
- `'Exponent`
- `'First`
- `'Floor`

* 'Fore
* 'Fraction
* 'Last
* 'Leading_Part
* 'Length
* 'Machine
* 'Machine_Emax
* 'Machine_Emin
* 'Machine_Mantissa
* 'Machine_Overflows
* 'Machine_Radix
* 'Machine_Rounds
* 'Max
* 'Min
* 'Model
* 'Model_Emin
* 'Model_Epsilon
* 'Model_Mantissa
* 'Model_Small
* 'Modulus
* 'Pos
* 'Pred
* 'Range
* 'Remainder
* 'Rounding
* 'Safe_First
* 'Safe_Last
* 'Scaling
* 'Signed_Zeros
* 'Size
* 'Small
* 'Succ
* 'Truncation
* 'Unbiased_Rounding
* 'Val
* 'Valid

This rule has no parameters.

Example

```
type Integer_A is access all Integer;

Var : aliased Integer := 1;
Var_A : Integer_A := Var'Access;  --  FLAG
```

9.4.5 `Non_Tagged_Derived_Types`

Flag all derived type declarations that do not have a record extension part.

This rule has no parameters.

Example

```
type Coordinates is record
   X, Y, Z : Float;
end record;

type Hidden_Coordinates is new Coordinates;   --  FLAG
```

9.4.6 `Outer_Loop_Exits`

Flag each `exit` statement containing a loop name that is not the name of the immediately enclosing `loop` statement.

This rule has no parameters.

Example

```
Outer : for J in S1'Range loop
   for K in S2'Range loop
      if S1 (J) = S2 (K) then
         Detected := True;
         exit Outer;                     --  FLAG
      end if;
   end loop;
end loop Outer;
```

9.4.7 `Overloaded_Operators`

Flag each function declaration that overloads an operator symbol. A function body is checked only if the body does not have a separate spec. Formal functions are also checked. For a renaming declaration, only renaming-as-declaration is checked

This rule has no parameters.

Example

```
type Rec is record
   C1 : Integer;
   C2 : Float;
end record;

function "<" (Left, Right : Rec) return Boolean;     --  FLAG
```

9.4.8 `Slices`

Flag all uses of array slicing

This rule has no parameters.

Example

```
procedure Proc (S : in out String; L, R : Positive) is
   Tmp : String := S (L .. R);         --  FLAG
begin
```

9.4.9 `Universal_Ranges`

Flag discrete ranges that are a part of an index constraint, constrained array definition, or `for`-loop parameter specification, and whose bounds are both of type *universal_integer*. Ranges that have at least one bound of a specific type (such as `1 .. N`, where `N` is a variable or an expression of non-universal type) are not flagged.

This rule has no parameters.

Example

```
L : Positive := 1;

S1 : String (L .. 10);
S2 : String (1 .. 10);     --  FLAG
```

This page is intentionally left blank.

CHAPTER

TEN

EXAMPLE OF *GNATCHECK* USAGE

Here is a simple example. Suppose that in the current directory we have a project file named `gnatcheck_example.gpr` with the following content:

```
project Gnatcheck_Example is

   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("main.adb");

   package Check is
      for Default_Switches ("ada") use ("-rules", "-from=coding_standard");
   end Check;

end Gnatcheck_Example;
```

And the file named `coding_standard` is also located in the current directory and has the following content:

```
-----------------------------------------------------
-- This is a sample gnatcheck coding standard file --
-----------------------------------------------------

--  First, turning on rules, that are directly implemented in gnatcheck
+RAbstract_Type_Declarations
+RAnonymous_Arrays
+RLocal_Packages
+RFloat_Equality_Checks
+REXIT_Statements_With_No_Loop_Name

--  Then, activating compiler checks of interest:
+RStyle_Checks:e
--  This style check checks if a unit name is present on END keyword that
--  is the end of the unit declaration
```

And the subdirectory `src` contains the following Ada sources:

`pack.ads`:

```
package Pack is
   type T is abstract tagged private;
   procedure P (X : T) is abstract;

   package Inner is
      type My_Float is digits 8;
      function Is_Equal (L, R : My_Float) return Boolean;
   end Inner;
private
   type T is abstract tagged null record;
end;
```

pack.adb:

```
package body Pack is
   package body Inner is
      function Is_Equal (L, R : My_Float) return Boolean is
      begin
         return L = R;
      end;
   end Inner;
end Pack;
```

and main.adb

```
with Pack; use Pack;
procedure Main is

   pragma Annotate
     (gnatcheck, Exempt_On, "Anonymous_Arrays", "this one is fine");
   Float_Array : array (1 .. 10) of Inner.My_Float;
   pragma Annotate (gnatcheck, Exempt_Off, "Anonymous_Arrays");

   Another_Float_Array : array (1 .. 10) of Inner.My_Float;

   use Inner;

   B : Boolean := False;

begin
   for J in Float_Array'Range loop
      if Is_Equal (Float_Array (J), Another_Float_Array (J)) then
         B := True;
         exit;
      end if;
   end loop;
end Main;
```

And suppose we call *gnatcheck* from the current directory using the project file as the only parameter of the call:

```
gnatcheck -Pgnatcheck_example.gpr
```

As a result, *gnatcheck* is called to check all the files from the project gnatcheck_example.gpr using the coding standard defined by the file coding_standard. The *gnatcheck* report file named gnatcheck.out will be created in the obj directory, and it will have the following content:

```
RULE CHECKING REPORT

1. OVERVIEW

Date and time of execution: 2009.10.28 14:17
```

```
Tool version: GNATCHECK (built with ASIS 2.0.R for GNAT Pro 6.3.0w (20091016))
Command line:

gnatcheck -files=... -cargs -gnatec=... -rules -from=coding_standard

Coding standard (applied rules):
   Abstract_Type_Declarations
   Anonymous_Arrays
   EXIT_Statements_With_No_Loop_Name
   Float_Equality_Checks
   Local_Packages

   Compiler style checks: -gnatye

Number of coding standard violations: 6
Number of exempted coding standard violations: 1

2. DETECTED RULE VIOLATIONS

2.1. NON-EXEMPTED VIOLATIONS

Source files with non-exempted violations
   pack.ads
   pack.adb
   main.adb

List of violations grouped by files, and ordered by increasing source location:

pack.ads:2:4: declaration of abstract type
pack.ads:5:4: declaration of local package
pack.ads:10:30: declaration of abstract type
pack.ads:11:1: (style) "end Pack" required
pack.adb:5:19: use of equality operation for float values
pack.adb:6:7: (style) "end Is_Equal" required
main.adb:9:26: anonymous array type
main.adb:19:10: exit statement with no loop name

2.2. EXEMPTED VIOLATIONS

Source files with exempted violations
   main.adb

List of violations grouped by files, and ordered by increasing source location:

main.adb:6:18: anonymous array type
   (this one is fine)

2.3. SOURCE FILES WITH NO VIOLATION

   No files without violations

END OF REPORT
```

This page is intentionally left blank.

CHAPTER

ELEVEN

LIST OF RULES

This section contains an alphabetized list of all the predefined GNATcheck rules.

- *Abort_Statements*
- *Abstract_Type_Declarations*
- *Address_Specifications_For_Initialized_Objects*
- *Address_Specifications_For_Local_Objects*
- *Anonymous_Arrays*
- *Anonymous_Subtypes*
- *Binary_Case_Statements*
- *Bit_Records_Without_Layout_Definition*
- *Blocks*
- *Boolean_Relational_Operators*
- *Complex_Inlined_Subprograms*
- *Conditional_Expressions*
- *Constructors*
- *Controlled_Type_Declarations*
- *Declarations_In_Blocks*
- *Deep_Inheritance_Hierarchies*
- *Deep_Library_Hierarchy*
- *Deeply_Nested_Generics*
- *Deeply_Nested_Inlining*
- *Default_Parameters*
- *Default_Values_For_Record_Components*
- *Deriving_From_Predefined_Type*
- *Direct_Calls_To_Primitives*
- *Discriminated_Records*
- *Downward_View_Conversions*
- *Enumeration_Ranges_In_CASE_Statements*

- *Enumeration_Representation_Clauses*
- *Exceptions_As_Control_Flow*
- *Exits_From_Conditional_Loops*
- *EXIT_Statements_With_No_Loop_Name*
- *Expanded_Loop_Exit_Names*
- *Explicit_Full_Discrete_Ranges*
- *Expression_Functions*
- *Fixed_Equality_Checks*
- *Float_Equality_Checks*
- *Forbidden_Attributes*
- *Forbidden_Pragmas*
- *Function_Style_Procedures*
- *Generics_In_Subprograms*
- *GOTO_Statements*
- *Implicit_IN_Mode_Parameters*
- *Implicit_SMALL_For_Fixed_Point_Types*
- *Improperly_Located_Instantiations*
- *Improper_Returns*
- *Incomplete_Representation_Specifications*
- *Maximum_Parameters*
- *Library_Level_Subprograms*
- *Local_Packages*
- *Local_USE_Clauses*
- *Metrics_Cyclomatic_Complexity*
- *Metrics_Essential_Complexity*
- *Metrics_LSLOC*
- *Misnamed_Controlling_Parameters*
- *Identifier_Suffixes*
- *Max_Identifier_Length*
- *Membership_Tests*
- *Misplaced_Representation_Items*
- *Multiple_Entries_In_Protected_Definitions*
- *Name_Clashes*
- *Nested_Subprograms*
- *No_Explicit_Real_Range*
- *No_Inherited_Classwide_Pre*

- *No_Scalar_Storage_Order_Specified*
- *Non_Qualified_Aggregates*
- *Non_Short_Circuit_Operators*
- *Non_SPARK_Attributes*
- *Non_Tagged_Derived_Types*
- *Non_Visible_Exceptions*
- *Number_Declarations*
- *Null_Paths*
- *Numeric_Indexing*
- *Numeric_Literals*
- *Object_Declarations_Out_Of_Order*
- *Objects_Of_Anonymous_Types*
- *One_Construct_Per_Line*
- *OTHERS_In_Aggregates*
- *OTHERS_In_CASE_Statements*
- *OTHERS_In_Exception_Handlers*
- *Outbound_Protected_Assignments*
- *Outer_Loop_Exits*
- *Overloaded_Operators*
- *Overly_Nested_Control_Structures*
- *Parameters_Out_Of_Order*
- *POS_On_Enumeration_Types*
- *Positional_Actuals_For_Defaulted_Generic_Parameters*
- *Positional_Actuals_For_Defaulted_Parameters*
- *Positional_Components*
- *Positional_Generic_Parameters*
- *Positional_Parameters*
- *Predicate_Testing*
- *Predefined_Numeric_Types*
- *Printable_ASCII*
- *Relative_Delay_Statements*
- *Representation_Specifications*
- *Quantified_Expressions*
- *Raising_External_Exceptions*
- *Raising_Predefined_Exceptions*
- *Separate_Numeric_Error_Handlers*

- *Single_Value_Enumeration_Types*
- *Slices*
- *Specific_Parent_Type_Invariant*
- *Specific_Pre_Post*
- *Specific_Type_Invariants*
- *Subprogram_Access*
- *Too_Many_Dependencies*
- *Too_Many_Primitives*
- *Too_Many_Parents*
- *Unassigned_OUT_Parameters*
- *Uncommented_BEGIN_In_Package_Bodies*
- *Recursive_Subprograms*
- *Unchecked_Address_Conversions*
- *Unchecked_Conversions_As_Actuals*
- *Unconditional_Exits*
- *Unconstrained_Array_Returns*
- *Unconstrained_Arrays*
- *Uninitialized_Global_Variables*
- *Universal_Ranges*
- *Unnamed_Blocks_And_Loops*
- *USE_PACKAGE_Clauses*
- *Visible_Components*
- *Volatile_Objects_Without_Address_Clauses*

APPENDIX A

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Copyright 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX