GNATcheck Reference Manual

Predefined Rules

GNAT, The GNU Ada Compiler
GNAT Version 4.6
Configuration level: 150355
Date: 2009/10/06

AdaCore

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 Guide Contains

This guide contains a description of each predefined `gnatcheck` rule, organized into categories.

- Chapter 1 [Style-Related Rules], page 3, presents the rules related to general programming style.
- Chapter 2 [Feature Usage Rules], page 17, presents rules that enforce usage patterns for specific features.
- Chapter 3 [Metrics-Related Rules], page 23, presents rules that enforce compliance with metric thresholds.
- Chapter 4 [SPARK Ada Rules], page 25, presents rules that enforce adherence to the Ada subset handled by the SPARK tools.
- Appendix A [List of Rules], page 29, 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 Guide

You should be familiar with the Ada language and with the usage of GNAT in general and with the `gnatcheck` tool in particular; please refer to the *GNAT User's Guide*.

1 Style-Related Rules

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

1.1 Tasking

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

1.1.1 `Multiple_Entries_In_Protected_Definitions`

Flag each protected definition (i.e., each protected object/type declaration) that defines 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.

1.1.2 `Volatile_Objects_Without_Address_Clauses`

Flag each volatile object that does not have an address clause.

The following check is made: if the pragma `Volatile` is applied to a data object or to its type, then an address clause must be supplied for this object.

This rule does not check the components of data objects, array components that are volatile as a result of the pragma `Volatile_Components`, or objects that are volatile because they are atomic as a result of pragmas `Atomic` or `Atomic_Components`.

Only variable declarations, and not constant declarations, are checked.

This rule has no parameters.

1.2 Object Orientation

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

1.2.1 `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.

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.

This rule does not flag private extension declarations. In the case of a private extension, the corresponding full declaration is checked.

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.

1.2.2 Direct_Calls_To_Primitives

Flags any non-dispatching call to a dispatching primitive operation, except for two cases:

- the common idiom where a primitive subprogram for a tagged type directly calls the same primitive subprogram of the type's immediate ancestor;
- the type for that the called operation is a primitive operation is declared as private non tagged type (with the full view tagged), and at the place of the call the full declaration of the type is not visible;

This rule has no parameters.

1.2.3 Too_Many_Parents

Flags any type declaration, single task declaration or single protected declaration that has more then 'N' parents, '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.

1.2.4 Visible_Components

Flags 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 type is considered as declaring a visible component if it contains a record definition by its own or as a part of a record extension. Type declaration is flagged even if it contains a record definition that defines no components.

Declarations located in private parts of local (generic) packages are not flagged. Declarations in private packages are not flagged.

This rule has no parameters.

1.3 Portability

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

1.3.1 `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 Section "Implementation Defined Attributes" in *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 Section "Implementation Defined Attributes" in *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 then one parameter is given in the same rule option, the parameters must be separated by commas.

If more then 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.

1.3.2 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 Section "Implementation Defined Pragmas" in *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 Section "Implementation Defined Pragmas" in *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 then one parameter is given in the same rule option, the parameters must be separated by a comma.

If more then 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.

1.3.3 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.

1.3.4 Predefined_Numeric_Types

Flag each explicit use of the name of any numeric type or subtype defined 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 over-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 defined in Standard. The use of numeric types and subtypes defined in other predefined packages (such as System.Any_Priority or Ada.Text_IO.Count) is not flagged

This rule has no parameters.

1.3.5 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.

1.4 Program Structure

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

1.4.1 Deeply_Nested_Generics

Flags a generic declaration nested in another generic declaration if the nesting level of the inner generic exceeds a 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 Positive integer specifying the maximal allowed nesting level for a generic declaration.

1.4.2 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.

1.4.3 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.

1.4.4 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.

1.5 Programming Practice

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

1.5.1 Anonymous_Arrays

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

This rule has no parameters.

1.5.2 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.

1.5.3 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.

1.5.4 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.

1.5.5 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.

1.5.6 GOTO_Statements

Flag each occurrence of a `goto` statement.

This rule has no parameters.

1.5.7 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.

1.5.8 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.

1.5.9 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.

1.5.10 OTHERS_In_CASE_Statements

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

This rule has no parameters.

1.5.11 OTHERS_In_Exception_Handlers

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

This rule has no parameters.

1.5.12 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
- 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 then one option is specified for the gnatcheck call, the later option and new parameter override the previous one(s).

1.5.13 `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.

1.5.14 `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.

1.5.15 `Positional_Components`

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

This rule has no parameters.

1.5.16 `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.

1.5.17 `Positional_Parameters`

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

- Parameters of calls to of prefix or infix operators 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 no parameters.

1.5.18 `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 has no parameters.

1.5.19 `Unconditional_Exits`

Flag unconditional `exit` statements.

This rule has no parameters.

1.5.20 `Unnamed_Blocks_And_Loops`

Flag each unnamed block statement and loop statement.

The rule has no parameters.

1.5.21 `USE_PACKAGE_Clauses`

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

This rule has no parameters.

1.6 Readability

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

1.6.1 `Misnamed_Controlling_Parameters`

Flags 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 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.

1.6.2 `Misnamed_Identifiers`

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
- constant declarations (but not number declarations)
- package renaming declarations (but not generic package renaming declarations)

This rule may have parameters. When used without parameters, the rule enforces the following checks:

- 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`

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.

The rule may have the following parameters:

- For the '`+R`' option:

  `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.

- 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=`*string*
: 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.

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

The '`-RMisnamed_Identifiers`' 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.

1.6.3 `Name_Clashes`

Check that certain names are not used as defining identifiers. To activate this rule, you need to supply a reference to the dictionary file(s) as a rule parameter(s) (more then one dictionary file can be specified). If no dictionary file is set, this rule will not cause anything to be flagged. Only defining occurrences, not references, are checked. The check is not case-sensitive.

This rule is enabled by default, but without setting any corresponding dictionary file(s); thus the default effect is to do no checks.

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

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.

1.6.4 `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.

1.7 Source Code Presentation

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

2 Feature Usage Rules

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

2.1 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.

2.2 Anonymous_Subtypes

Flag all uses of anonymous subtypes (except cases when subtype indication is a part of a record component definition, and this subtype indication depends on a discriminant). 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.

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

```
for I in 1 .. N loop
   ...
end loop;
```

Declaring an explicit subtype solves the problem:

```
subtype S is Integer range 1..N;
...
for I in S loop
   ...
end loop;
```

This rule has no parameters.

2.3 Blocks

Flag each block statement.

This rule has no parameters.

2.4 Complex_Inlined_Subprograms

Flags a subprogram (or generic subprogram) if pragma Inline is applied to the subprogram 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 a if statement, or a short circuit control form;
- the number of statements exceeds a value specified by the 'N' rule parameter;

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.

2.5 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.

2.6 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.

2.7 Deeply_Nested_Inlining

Flags a subprogram (or generic subprogram) if pragma Inline has been applied to the subprogram but the subprogram calls to another inlined subprogram that results in nested inlining with 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 maximal allowed level of nested inlining.

2.8 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.

2.9 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.

2.10 Explicit_Full_Discrete_Ranges

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

This rule has no parameters.

2.11 Float_Equality_Checks

Flag all calls to the predefined equality operations for floating-point types. Both "`=`" and "`/=`" operations are checked. User-defined equality operations are not flagged, nor are "`=`" and "`/=`" operations for fixed-point types.

This rule has no parameters.

2.12 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.

2.13 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.

2.14 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.

2.15 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.

2.16 Library_Level_Subprograms

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

This rule has no parameters.

2.17 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.

2.18 Numeric_Literals

Flag each use of a numeric literal in an index expression, and in any circumstance except for the following:

- a literal occurring in the initialization expression for a constant declaration or a named number declaration, or
- an integer literal that is less than or equal to a value specified by the '`N`' rule parameter.

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

If no parameters are set, the maximum unflagged value is 1.

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 but retains the last specified maximum unflagged value. If the '+R' option subsequently appears, this value is used as the threshold for the check.

2.19 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.

2.20 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.

2.21 Unassigned_OUT_Parameters

Flags procedures' `out` parameters that are not assigned, and identifies the contexts in which the assignments are missing.

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

An `out` parameter is flagged in an exception handler in the exception part of the procedure body's handled sequence of statements if the handler contains neither 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.

This rule does not have any parameters.

2.22 `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 checked. Function calls and function renamings are not checked.

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

This rule has no parameters.

3 Metrics-Related Rules

The rules in this chapter 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
```

3.1 `Metrics_Essential_Complexity`

The `Metrics_Essential_Complexity` rule takes a positive integer as upper bound. A construct exceeding this limit will be flagged.

3.2 `Metrics_Cyclomatic_Complexity`

The `Metrics_Cyclomatic_Complexity` rule takes a positive integer as upper bound. A construct exceeding this limit will be flagged.

3.3 `Metrics_LSLOC`

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

4 SPARK Ada Rules

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

4.1 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.

4.2 Expanded_Loop_Exit_Names

Flag all expanded loop names in `exit` statements.

This rule has no parameters.

4.3 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.

4.4 Non_Tagged_Derived_Types

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

This rule has no parameters.

4.5 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.

4.6 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.

4.7 Slices

Flag all uses of array slicing

This rule has no parameters.

4.8 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.

Appendix A List of Rules

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

- `Abstract_Type_Declarations`
  See Section 2.1 [Abstract_Type_Declarations], page 17.
- `Anonymous_Arrays`
  See Section 1.5.1 [Anonymous_Arrays], page 9.
- `Anonymous_Subtypes`
  See Section 2.2 [Anonymous_Subtypes], page 17.
- `Blocks`
  See Section 2.3 [Blocks], page 17.
- `Boolean_Relational_Operators`
  See Section 4.1 [Boolean_Relational_Operators], page 25.
- `Complex_Inlined_Subprograms`
  See Section 2.4 [Complex_Inlined_Subprograms], page 17.
- `Controlled_Type_Declarations`
  See Section 2.5 [Controlled_Type_Declarations], page 18.
- `Declarations_In_Blocks`
  See Section 2.6 [Declarations_In_Blocks], page 18.
- `Deep_Inheritance_Hierarchies`
  See Section 1.2.1 [Deep_Inheritance_Hierarchies], page 3.
- `Deeply_Nested_Generics`
  See Section 1.4.1 [Deeply_Nested_Generics], page 8.
- `Deeply_Nested_Inlining`
  See Section 2.7 [Deeply_Nested_Inlining], page 18.
- `Default_Parameters`
  See Section 2.8 [Default_Parameters], page 18.
- `Direct_Calls_To_Primitives`
  See Section 1.2.2 [Direct_Calls_To_Primitives], page 4.
- `Discriminated_Records`
  See Section 2.9 [Discriminated_Records], page 19.
- `Enumeration_Ranges_In_CASE_Statements`
  See Section 1.5.2 [Enumeration_Ranges_In_CASE_Statements], page 9.
- `Exceptions_As_Control_Flow`
  See Section 1.5.3 [Exceptions_As_Control_Flow], page 9.
- `Exits_From_Conditional_Loops`
  See Section 1.5.4 [Exits_From_Conditional_Loops], page 9.

- `EXIT_Statements_With_No_Loop_Name`
  See Section 1.5.5 [EXIT_Statements_With_No_Loop_Name], page 9.
- `Expanded_Loop_Exit_Names`
  See Section 4.2 [Expanded_Loop_Exit_Names], page 25.
- `Explicit_Full_Discrete_Ranges`
  See Section 2.10 [Explicit_Full_Discrete_Ranges], page 19.
- `Float_Equality_Checks`
  See Section 2.11 [Float_Equality_Checks], page 19.
- `Forbidden_Attributes`
  See Section 1.3.1 [Forbidden_Attributes], page 5.
- `Forbidden_Pragmas`
  See Section 1.3.2 [Forbidden_Pragmas], page 6.
- `Function_Style_Procedures`
  See Section 2.12 [Function_Style_Procedures], page 19.
- `Generics_In_Subprograms`
  See Section 2.13 [Generics_In_Subprograms], page 19.
- `GOTO_Statements`
  See Section 1.5.6 [GOTO_Statements], page 9.
- `Implicit_IN_Mode_Parameters`
  See Section 2.14 [Implicit_IN_Mode_Parameters], page 20.
- `Implicit_SMALL_For_Fixed_Point_Types`
  See Section 1.3.3 [Implicit_SMALL_For_Fixed_Point_Types], page 7.
- `Improperly_Located_Instantiations`
  See Section 2.15 [Improperly_Located_Instantiations], page 20.
- `Improper_Returns`
  See Section 1.5.7 [Improper_Returns], page 10.
- `Library_Level_Subprograms`
  See Section 2.16 [Library_Level_Subprograms], page 20.
- `Local_Packages`
  See Section 1.4.2 [Local_Packages], page 8.
- `Metrics_Cyclomatic_Complexity`
  See Section 3.2 [Metrics_Cyclomatic_Complexity], page 23.
- `Metrics_Essential_Complexity`
  See Section 3.1 [Metrics_Essential_Complexity], page 23.
- `Metrics_LSLOC`
  See Section 3.3 [Metrics_LSLOC], page 23.
- `Misnamed_Controlling_Parameters`
  See Section 1.6.1 [Misnamed_Controlling_Parameters], page 13.

- `Misnamed_Identifiers`
  See Section 1.6.2 [Misnamed_Identifiers], page 13.
- `Multiple_Entries_In_Protected_Definitions`
  See Section 1.1.1 [Multiple_Entries_In_Protected_Definitions], page 3.
- `Name_Clashes`
  See Section 1.6.3 [Name_Clashes], page 15.
- `Non_Qualified_Aggregates`
  See Section 2.17 [Non_Qualified_Aggregates], page 20.
- `Non_Short_Circuit_Operators`
  See Section 1.5.8 [Non_Short_Circuit_Operators], page 10.
- `Non_SPARK_Attributes`
  See Section 4.3 [Non_SPARK_Attributes], page 25.
- `Non_Tagged_Derived_Types`
  See Section 4.4 [Non_Tagged_Derived_Types], page 27.
- `Non_Visible_Exceptions`
  See Section 1.4.3 [Non_Visible_Exceptions], page 8.
- `Numeric_Literals`
  See Section 2.18 [Numeric_Literals], page 20.
- `OTHERS_In_Aggregates`
  See Section 1.5.9 [OTHERS_In_Aggregates], page 10.
- `OTHERS_In_CASE_Statements`
  See Section 1.5.10 [OTHERS_In_CASE_Statements], page 10.
- `OTHERS_In_Exception_Handlers`
  See Section 1.5.11 [OTHERS_In_Exception_Handlers], page 10.
- `Outer_Loop_Exits`
  See Section 4.5 [Outer_Loop_Exits], page 27.
- `Overloaded_Operators`
  See Section 4.6 [Overloaded_Operators], page 27.
- `Overly_Nested_Control_Structures`
  See Section 1.5.12 [Overly_Nested_Control_Structures], page 10.
- `Parameters_Out_Of_Order`
  See Section 2.19 [Parameters_Out_Of_Order], page 21.
- `Positional_Actuals_For_Defaulted_Generic_Parameters`
  See Section 1.5.13 [Positional_Actuals_For_Defaulted_Generic_Parameters], page 11.
- `Positional_Actuals_For_Defaulted_Parameters`
  See Section 1.5.14 [Positional_Actuals_For_Defaulted_Parameters], page 11.

- `Positional_Components`
  See Section 1.5.15 [Positional_Components], page 11.
- `Positional_Generic_Parameters`
  See Section 1.5.16 [Positional_Generic_Parameters], page 11.
- `Positional_Parameters`
  See Section 1.5.17 [Positional_Parameters], page 12.
- `Predefined_Numeric_Types`
  See Section 1.3.4 [Predefined_Numeric_Types], page 7.
- `Raising_External_Exceptions`
  See Section 1.4.4 [Raising_External_Exceptions], page 8.
- `Raising_Predefined_Exceptions`
  See Section 2.20 [Raising_Predefined_Exceptions], page 21.
- `Separate_Numeric_Error_Handlers`
  See Section 1.3.5 [Separate_Numeric_Error_Handlers], page 7.
- `Slices`
  See Section 4.7 [Slices], page 27.
- `Too_Many_Parents`
  See Section 1.2.3 [Too_Many_Parents], page 4.
- `Unassigned_OUT_Parameters`
  See Section 2.21 [Unassigned_OUT_Parameters], page 21.
- `Uncommented_BEGIN_In_Package_Bodies`
  See Section 1.6.4 [Uncommented_BEGIN_In_Package_Bodies], page 16.
- `Recursive_Subprograms`
  See Section 1.5.18 [Recursive_Subprograms], page 12.
- `Unconditional_Exits`
  See Section 1.5.19 [Unconditional_Exits], page 12.
- `Unconstrained_Array_Returns`
  See Section 2.22 [Unconstrained_Array_Returns], page 22.
- `Universal_Ranges`
  See Section 4.8 [Universal_Ranges], page 27.
- `Unnamed_Blocks_And_Loops`
  See Section 1.5.20 [Unnamed_Blocks_And_Loops], page 12.
- `USE_PACKAGE_Clauses`
  See Section 1.5.21 [USE_PACKAGE_Clauses], page 12.
- `Visible_Components`
  See Section 1.2.4 [Visible_Components], page 4.
- `Volatile_Objects_Without_Address_Clauses`
  See Section 1.1.2 [Volatile_Objects_Without_Address_Clauses], page 3.

Index

P

R

S

T

U

V

Table of Contents