

GNAT Reference Manual

GNAT Reference Manual , Jan 09, 2026

AdaCore

Copyright © 2008-2026, Free Software Foundation

2.38	Pragma CPP_Class	23
2.39	Pragma CPP_Constructor	23
2.40	Pragma CPP_Virtual	24
2.41	Pragma CPP_Vtable	24
2.42	Pragma CPU	24
2.43	Pragma Deadline_Floor	24
2.44	Pragma Debug	24
2.45	Pragma Debug_Policy	25
2.46	Pragma Default_Initial_Condition	25
2.47	Pragma Default_Scalar_Storage_Order	25
2.48	Pragma Default_Storage_Pool	26
2.49	Pragma Depends	26
2.50	Pragma Detect_Blocking	27
2.51	Pragma Disable_Atomic_Synchronization	27
2.52	Pragma Dispatching_Domain	28
2.53	Pragma Effective_Reads	28
2.54	Pragma Effective_Writes	28
2.55	Pragma Elaboration_Checks	28
2.56	Pragma Eliminate	28
2.57	Pragma Enable_Atomic_Synchronization	31
2.58	Pragma Exceptional_Cases	31
2.59	Pragma Exit_Cases	31
2.60	Pragma Export_Function	31
2.61	Pragma Export_Object	32
2.62	Pragma Export_Procedure	33
2.63	Pragma Export_Valued_Procedure	34
2.64	Pragma Extend_System	34
2.65	Pragma Extensions_Allowed	35
2.66	Pragma Extensions_Visible	35
2.67	Pragma External	36
2.68	Pragma External_Name_Casing	36
2.69	Pragma Fast_Math	37
2.70	Pragma Favor_Top_Level	37
2.71	Pragma Finalize_Storage_Only	37
2.72	Pragma Float_Representation	38
2.73	Pragma Ghost	38
2.74	Pragma Global	38
2.75	Pragma Ident	39
2.76	Pragma Ignore_Pragma	39
2.77	Pragma Implementation_Defined	39
2.78	Pragma Implemented	39
2.79	Pragma Implicit_Packing	40
2.80	Pragma Import_Function	41
2.81	Pragma Import_Object	42
2.82	Pragma Import_Procedure	42
2.83	Pragma Import_Valued_Procedure	43
2.84	Pragma Independent	44

‘GNAT, The GNU Ada Development Environment’

GCC version 16.0.1

AdaCore

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, with the Front-Cover Texts being “GNAT Reference Manual”, and with no Back-Cover Texts. A copy of the license is included in the section entitled [GNU Free Documentation License], page 381.

1 About This Guide

This manual contains useful information in writing programs using the GNAT compiler. It includes information on implementation dependent characteristics of GNAT, including all the information required by Annex M of the Ada language standard.

GNAT implements Ada 95, Ada 2005, Ada 2012 and Ada 2022, and it may also be invoked in Ada 83 compatibility mode. By default, GNAT assumes Ada 2012, but you can override with a compiler switch to explicitly specify the language version. (Please refer to the ‘GNAT User’s Guide’ for details on these switches.) Throughout this manual, references to ‘Ada’ without a year suffix apply to all the Ada versions of the language.

Ada is designed to be highly portable. In general, a program will have the same effect even when compiled by different compilers on different platforms. However, since Ada is designed to be used in a wide variety of applications, it also contains a number of system dependent features to be used in interfacing to the external world.

Note: Any program that makes use of implementation-dependent features may be non-portable. You should follow good programming practice and isolate and clearly document any sections of your program that make use of these features in a non-portable manner.

1.1 What This Reference Manual Contains

This reference manual contains the following chapters:

- * [Implementation Defined Pragmas], page 4, lists GNAT implementation-dependent pragmas, which can be used to extend and enhance the functionality of the compiler.
- * [Implementation Defined Attributes], page 120, lists GNAT implementation-dependent attributes, which can be used to extend and enhance the functionality of the compiler.
- * [Standard and Implementation Defined Restrictions], page 144, lists GNAT implementation-dependent restrictions, which can be used to extend and enhance the functionality of the compiler.
- * [Implementation Advice], page 159, provides information on generally desirable behavior which are not requirements that all compilers must follow since it cannot be provided on all systems, or which may be undesirable on some systems.
- * [Implementation Defined Characteristics], page 179, provides a guide to minimizing implementation dependent features.
- * [Intrinsic Subprograms], page 198, describes the intrinsic subprograms implemented by GNAT, and how they can be imported into user application programs.
- * [Representation Clauses and Pragmas], page 201, describes in detail the way that GNAT represents data, and in particular the exact set of representation clauses and pragmas that is accepted.
- * [Standard Library Routines], page 232, provides a listing of packages and a brief description of the functionality that is provided by Ada’s extensive set of standard library routines as implemented by GNAT.
- * [The Implementation of Standard I/O], page 243, details how the GNAT implementation of the input-output facilities.
- * [The GNAT Library], page 261, is a catalog of packages that complement the Ada predefined library.

- * [Interfacing to Other Languages], page 279, describes how programs written in Ada using GNAT can be interfaced to other programming languages.
- * [Specialized Needs Annexes], page 285, describes the GNAT implementation of all of the specialized needs annexes.
- * [Implementation of Specific Ada Features], page 286, discusses issues related to GNAT's implementation of machine code insertions, tasking, and several other features.
- * [Implementation of Ada 2022 Features], page 296, describes the status of the GNAT implementation of the Ada 2022 language standard.
- * [Security Hardening Features], page 362, documents GNAT extensions aimed at security hardening.
- * [Obsolescent Features], page 370, documents implementation dependent features, including pragmas and attributes, which are considered obsolescent, since there are other preferred ways of achieving the same results. These obsolescent forms are retained for backwards compatibility.
- * [Compatibility and Porting Guide], page 372, presents some guidelines for developing portable Ada code, describes the compatibility issues that may arise between GNAT and other Ada compilation systems (including those for Ada 83), and shows how GNAT can expedite porting applications developed in other Ada environments.
- * [GNU Free Documentation License], page 381, contains the license for this document.

This reference manual assumes a basic familiarity with the Ada 95 language, as described in the *International Standard ANSI/ISO/IEC-8652:1995*. It does not require knowledge of the new features introduced by Ada 2005 or Ada 2012. All three reference manuals are included in the GNAT documentation package.

1.2 Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- * **Functions, utility program names, standard names, and classes.**
- * **Option flags**
- * **File names**
- * **Variables**
- * ‘**Emphasis**’
- * [optional information or parameters]
- * Examples are described by text
 and then shown this way.
- * Commands that are entered by the user are shown as preceded by a prompt string comprising the \$ character followed by a space.

1.3 Related Information

See the following documents for further information on GNAT:

- * *GNAT User's Guide for Native Platforms*, which provides information on how to use the GNAT development environment.

- * *Ada 95 Reference Manual*, the Ada 95 programming language standard.
- * *Ada 95 Annotated Reference Manual*, which is an annotated version of the Ada 95 standard. The annotations describe detailed aspects of the design decision, and in particular contain useful sections on Ada 83 compatibility.
- * *Ada 2005 Reference Manual*, the Ada 2005 programming language standard.
- * *Ada 2005 Annotated Reference Manual*, which is an annotated version of the Ada 2005 standard. The annotations describe detailed aspects of the design decision.
- * *Ada 2012 Reference Manual*, the Ada 2012 programming language standard.
- * *DEC Ada, Technical Overview and Comparison on DIGITAL Platforms*, which contains specific information on compatibility between GNAT and DEC Ada 83 systems.
- * *DEC Ada, Language Reference Manual*, part number AA-PYZAB-TK, which describes in detail the pragmas and attributes provided by the DEC Ada 83 compiler system.

2 Implementation Defined Pragmas

Ada defines a set of pragmas that can be used to supply additional information to the compiler. These language defined pragmas are implemented in GNAT and work as described in the Ada Reference Manual.

In addition, Ada allows implementations to define additional pragmas whose meaning is defined by the implementation. GNAT provides a number of these implementation-defined pragmas, which can be used to extend and enhance the functionality of the compiler. This section of the GNAT Reference Manual describes these additional pragmas.

Note that any program using these pragmas might not be portable to other compilers (although GNAT implements this set of pragmas on all platforms). Therefore if portability to other compilers is an important consideration, the use of these pragmas should be minimized.

2.1 Pragma Abort_Defer

Syntax:

```
pragma Abort_Defer;
```

This pragma must appear at the start of the statement sequence of a handled sequence of statements (right after the `begin`). It has the effect of deferring aborts for the sequence of statements (but not for the declarations or handlers, if any, associated with this statement sequence). This can also be useful for adding a polling point in Ada code, where asynchronous abort of tasks is checked when leaving the statement sequence, and is lighter than, for example, using `delay 0.0;`, since with zero-cost exception handling, propagating exceptions (implicitly used to implement task abort) cannot be done reliably in an asynchronous way.

An example of usage would be:

```
-- Add a polling point to check for task aborts

begin
  pragma Abort_Defer;
end;
```

2.2 Pragma Abstract_State

Syntax:

```
pragma Abstract_State (ABSTRACT_STATE_LIST);

ABSTRACT_STATE_LIST ::=
  null
  | STATE_NAME_WITH_OPTIONS
  | (STATE_NAME_WITH_OPTIONS {, STATE_NAME_WITH_OPTIONS} )

STATE_NAME_WITH_OPTIONS ::=
  STATE_NAME
  | (STATE_NAME with OPTION_LIST)
```

```

OPTION_LIST ::= OPTION {, OPTION}

OPTION ::=
    SIMPLE_OPTION
  | NAME_VALUE_OPTION

SIMPLE_OPTION ::= Ghost | Synchronous

NAME_VALUE_OPTION ::=
    Part_Of => ABSTRACT_STATE
  | External [=> EXTERNAL_PROPERTY_LIST]

EXTERNAL_PROPERTY_LIST ::=
    EXTERNAL_PROPERTY
  | (EXTERNAL_PROPERTY {, EXTERNAL_PROPERTY} )

EXTERNAL_PROPERTY ::=
    Async_Readers      [=> static_boolean_EXPRESSION]
  | Async_Writers      [=> static_boolean_EXPRESSION]
  | Effective_Reads    [=> static_boolean_EXPRESSION]
  | Effective_Writes   [=> static_boolean_EXPRESSION]
  | others              => static_boolean_EXPRESSION

STATE_NAME ::= defining_identifier

ABSTRACT_STATE ::= name

```

For the semantics of this pragma, see the entry for aspect `Abstract_State` in the SPARK 2014 Reference Manual, section 7.1.4.

2.3 Pragma `Ada_83`

Syntax:

```
pragma Ada_83;
```

A configuration pragma that establishes Ada 83 mode for the unit to which it applies, regardless of the mode set by the command line switches. In Ada 83 mode, GNAT attempts to be as compatible with the syntax and semantics of Ada 83, as defined in the original Ada 83 Reference Manual as possible. In particular, the keywords added by Ada 95 and Ada 2005 are not recognized, optional package bodies are allowed, and generics may name types with unknown discriminants without using the (<>) notation. In addition, some but not all of the additional restrictions of Ada 83 are enforced.

Like all configuration pragmas, if the pragma is placed before a library level package specification it is not propagated to the corresponding package body (see RM 10.1.5(8)); it must be added explicitly to the package body.

Ada 83 mode is intended for two purposes. Firstly, it allows existing Ada 83 code to be compiled and adapted to GNAT with less effort. Secondly, it aids in keeping code backwards

compatible with Ada 83. However, there is no guarantee that code that is processed correctly by GNAT in Ada 83 mode will in fact compile and execute with an Ada 83 compiler, since GNAT does not enforce all the additional checks required by Ada 83.

2.4 Pragma Ada_95

Syntax:

```
pragma Ada_95;
```

A configuration pragma that establishes Ada 95 mode for the unit to which it applies, regardless of the mode set by the command line switches. This mode is set automatically for the `Ada` and `System` packages and their children, so you need not specify it in these contexts. This pragma is useful when writing a reusable component that itself uses Ada 95 features, but which is intended to be usable from either Ada 83 or Ada 95 programs.

Like all configuration pragmas, if the pragma is placed before a library level package specification it is not propagated to the corresponding package body (see RM 10.1.5(8)); it must be added explicitly to the package body.

2.5 Pragma Ada_05

Syntax:

```
pragma Ada_05;  
pragma Ada_05 (local_NAME);
```

A configuration pragma that establishes Ada 2005 mode for the unit to which it applies, regardless of the mode set by the command line switches. This pragma is useful when writing a reusable component that itself uses Ada 2005 features, but which is intended to be usable from either Ada 83 or Ada 95 programs.

Like all configuration pragmas, if the pragma is placed before a library level package specification it is not propagated to the corresponding package body (see RM 10.1.5(8)); it must be added explicitly to the package body.

The one argument form (which is not a configuration pragma) is used for managing the transition from Ada 95 to Ada 2005 in the run-time library. If an entity is marked as `Ada_2005` only, then referencing the entity in `Ada_83` or `Ada_95` mode will generate a warning. In addition, in `Ada_83` or `Ada_95` mode, a preference rule is established which does not choose such an entity unless it is unambiguously specified. This avoids extra subprograms marked this way from generating ambiguities in otherwise legal pre-Ada_2005 programs. The one argument form is intended for exclusive use in the GNAT run-time library.

2.6 Pragma Ada_2005

Syntax:

```
pragma Ada_2005;
```

This configuration pragma is a synonym for `pragma Ada_05` and has the same syntax and effect.

2.7 Pragma Ada_12

Syntax:

```
pragma Ada_12;  
pragma Ada_12 (local_NAME);
```

A configuration pragma that establishes Ada 2012 mode for the unit to which it applies, regardless of the mode set by the command line switches. This mode is set automatically for the `Ada` and `System` packages and their children, so you need not specify it in these contexts. This pragma is useful when writing a reusable component that itself uses Ada 2012 features, but which is intended to be usable from Ada 83, Ada 95, or Ada 2005 programs.

Like all configuration pragmas, if the pragma is placed before a library level package specification it is not propagated to the corresponding package body (see RM 10.1.5(8)); it must be added explicitly to the package body.

The one argument form, which is not a configuration pragma, is used for managing the transition from Ada 2005 to Ada 2012 in the run-time library. If an entity is marked as `Ada_2012` only, then referencing the entity in any pre-`Ada_2012` mode will generate a warning. In addition, in any pre-`Ada_2012` mode, a preference rule is established which does not choose such an entity unless it is unambiguously specified. This avoids extra subprograms marked this way from generating ambiguities in otherwise legal pre-`Ada_2012` programs. The one argument form is intended for exclusive use in the GNAT run-time library.

2.8 Pragma Ada_2012

Syntax:

```
pragma Ada_2012;
```

This configuration pragma is a synonym for `pragma Ada_12` and has the same syntax and effect.

2.9 Pragma Ada_2022

Syntax:

```
pragma Ada_2022;  
pragma Ada_2022 (local_NAME);
```

A configuration pragma that establishes Ada 2022 mode for the unit to which it applies, regardless of the mode set by the command line switches. This mode is set automatically for the `Ada` and `System` packages and their children, so you need not specify it in these contexts. This pragma is useful when writing a reusable component that itself uses Ada 2022 features, but which is intended to be usable from Ada 83, Ada 95, Ada 2005 or Ada 2012 programs.

Like all configuration pragmas, if the pragma is placed before a library level package specification it is not propagated to the corresponding package body (see RM 10.1.5(8)); it must be added explicitly to the package body.

The one argument form, which is not a configuration pragma, is used for managing the transition from Ada 2012 to Ada 2022 in the run-time library. If an entity is marked as `Ada_2022` only, then referencing the entity in any pre-`Ada_2022` mode will generate a

warning. In addition, in any pre-Ada_2012 mode, a preference rule is established which does not choose such an entity unless it is unambiguously specified. This avoids extra subprograms marked this way from generating ambiguities in otherwise legal pre-Ada_2022 programs. The one argument form is intended for exclusive use in the GNAT run-time library.

2.10 Pragma `Aggregate_Individually_Assign`

Syntax:

```
pragma Aggregate_Individually_Assign;
```

Where possible, GNAT will store the binary representation of a record aggregate in memory for space and performance reasons. This configuration pragma changes this behavior so that record aggregates are instead always converted into individual assignment statements.

2.11 Pragma `Allow_Integer_Address`

Syntax:

```
pragma Allow_Integer_Address;
```

In almost all versions of GNAT, `System.Address` is a private type in accordance with the implementation advice in the RM. This means that integer values, in particular integer literals, are not allowed as address values. If the configuration pragma `Allow_Integer_Address` is given, then integer expressions may be used anywhere a value of type `System.Address` is required. The effect is to introduce an implicit unchecked conversion from the integer value to type `System.Address`. The reverse case of using an address where an integer type is required is handled analogously. The following example compiles without errors:

```
pragma Allow_Integer_Address;
with System; use System;
package AddrAsInt is
  X : Integer;
  Y : Integer;
  for X'Address use 16#1240#;
  for Y use at 16#3230#;
  m : Address := 16#4000#;
  n : constant Address := 4000;
  p : constant Address := Address (X + Y);
  v : Integer := y'Address;
  w : constant Integer := Integer (Y'Address);
  type R is new integer;
  RR : R := 1000;
  Z : Integer;
  for Z'Address use RR;
end AddrAsInt;
```

Note that pragma `Allow_Integer_Address` is ignored if `System.Address` is not a private type. In implementations of GNAT where `System.Address` is a visible integer type, this pragma serves no purpose but is ignored rather than rejected to allow common sets of sources to be used in the two situations.

2.12 Pragma Always_Terminates

Syntax:

```
pragma Always_Terminates [ (boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Always_Terminates` in the SPARK 2014 Reference Manual, section 6.1.11.

2.13 Pragma Annotate

Syntax:

```
pragma Annotate (IDENTIFIER [, IDENTIFIER {, ARG}] [, entity => local_NAME]);
```

```
ARG ::= NAME | EXPRESSION
```

This pragma is used to annotate programs. `IDENTIFIER` identifies the type of annotation. GNAT verifies that it is an identifier, but does not otherwise analyze it. The second optional identifier is also left unanalyzed, and by convention is used to control the action of the tool to which the annotation is addressed. The remaining `ARG` arguments can be either string literals or more generally expressions. String literals (and concatenations of string literals) are assumed to be either of type `Standard.String` or else `Wide_String` or `Wide_Wide_String` depending on the character literals they contain. All other kinds of arguments are analyzed as expressions, and must be unambiguous. The last argument if present must have the identifier `Entity` and GNAT verifies that a local name is given.

The analyzed pragma is retained in the tree, but not otherwise processed by any part of the GNAT compiler, except to generate corresponding note lines in the generated ALI file. For the format of these note lines, see the compiler source file `lib-writ.ads`. This pragma is intended for use by external tools. The use of pragma `Annotate` does not affect the compilation process in any way. This pragma may be used as a configuration pragma.

2.14 Pragma Assert

Syntax:

```
pragma Assert (
  boolean_EXPRESSION
  [, string_EXPRESSION]);
```

The effect of this pragma depends on whether the corresponding command line switch is set to activate assertions. The pragma expands into code equivalent to the following:

```
if assertions-enabled then
  if not boolean_EXPRESSION then
    System.Assertions.Raise_Assert_Failure
      (string_EXPRESSION);
  end if;
end if;
```

The string argument, if given, is the message that will be associated with the exception occurrence if the exception is raised. If no second argument is given, the default message is `file:nnn`, where `file` is the name of the source file containing the assert, and `nnn` is the line number of the assert.


```
[, depends => DEPENDENCY_DESCRIPTOR]);
```

```
DEPENDENCY_DESCRIPTOR ::= LEVEL_IDENTIFIER | LEVEL_IDENTIFIER_LIST
```

```
LEVEL_IDENTIFIER_LIST ::= '[' LEVEL_IDENTIFIER {, LEVEL_IDENTIFIER} ']'
```

For the semantics of this pragma, see the SPARK 2014 Reference Manual, section 11.4.3.

2.17 Pragma Assertion_Policy

Syntax:

```
pragma Assertion_Policy (CHECK | DISABLE | IGNORE | SUPPRESSIBLE);
```

```
pragma Assertion_Policy (
  ASSERTION_KIND => POLICY_IDENTIFIER
  {, ASSERTION_KIND => POLICY_IDENTIFIER});
```

```
ASSERTION_KIND ::= RM_ASSERTION_KIND | ID_ASSERTION_KIND | ASSERTION_LEVEL
```

```
RM_ASSERTION_KIND ::= Assert
                    Static_Predicate
                    Dynamic_Predicate
                    Pre
                    Pre'Class
                    Post
                    Post'Class
                    Type_Invariant
                    Type_Invariant'Class
                    Default_Initial_Condition
```

```
ID_ASSERTION_KIND ::= Assertions
                    Assert_And_Cut
                    Assume
                    Contract_Cases
                    Debug
                    Ghost
                    Initial_Condition
                    Invariant
                    Invariant'Class
                    Loop_Invariant
                    Loop_Variant
                    Postcondition
                    Precondition
                    Predicate
                    Refined_Post
                    Statement_Assertions
                    Subprogram_Variant
```


2.22 Pragma Attribute_Definition

Syntax:

```
pragma Attribute_Definition
  ([Attribute =>] ATTRIBUTE_DESIGNATOR,
   [Entity    =>] LOCAL_NAME,
   [Expression =>] EXPRESSION | NAME);
```

If **Attribute** is a known attribute name, this pragma is equivalent to the attribute definition clause:

```
for Entity'Attribute use Expression;
```

If **Attribute** is not a recognized attribute name, the pragma is ignored, and a warning is emitted. This allows source code to be written that takes advantage of some new attribute, while remaining compilable with earlier compilers.

2.23 Pragma C_Pass_By_Copy

Syntax:

```
pragma C_Pass_By_Copy
  ([Max_Size =>] static_integer_EXPRESSION);
```

Normally the default mechanism for passing C convention records to C convention subprograms is to pass them by reference, as suggested by RM B.3(69). Use the configuration pragma **C_Pass_By_Copy** to change this default, by requiring that record formal parameters be passed by copy if all of the following conditions are met:

- * The size of the record type does not exceed the value specified for **Max_Size**.
- * The record type has **Convention C**.
- * The formal parameter has this record type, and the subprogram has a foreign (non-Ada) convention.

If these conditions are met the argument is passed by copy; i.e., in a manner consistent with what C expects if the corresponding formal in the C prototype is a struct (rather than a pointer to a struct).

You can also pass records by copy by specifying the convention **C_Pass_By_Copy** for the record type, or by using the extended **Import** and **Export** pragmas, which allow specification of passing mechanisms on a parameter by parameter basis.

2.24 Pragma Check

Syntax:

```
pragma Check (
  [Name    =>] CHECK_KIND,
  [Check   =>] Boolean_EXPRESSION
  [, [Message =>] string_EXPRESSION] );
```

```
CHECK_KIND ::= IDENTIFIER      |
               Pre'Class       |
               Post'Class      |
```


This is a configuration pragma that defines a new implementation defined check name (unless IDENTIFIER matches one of the predefined check names, in which case the pragma has no effect). Check names are global to a partition, so if two or more configuration pragmas are present in a partition mentioning the same name, only one new check name is introduced.

An implementation defined check name introduced with this pragma may be used in only three contexts: `pragma Suppress`, `pragma Unsuppress`, and as the prefix of a `Check_Name'Enabled` attribute reference. For any of these three cases, the check name must be visible. A check name is visible if it is in the configuration pragmas applying to the current unit, or if it appears at the start of any unit that is part of the dependency set of the current unit (e.g., units that are mentioned in `with` clauses).

Check names introduced by this pragma are subject to control by compiler switches (in particular `-gnatp`) in the usual manner.

2.27 Pragma Check_Policy

Syntax:

```
pragma Check_Policy
  ([Name    =>] CHECK_KIND,
   [Policy =>] POLICY_IDENTIFIER);

pragma Check_Policy (
  CHECK_KIND => POLICY_IDENTIFIER
  {, CHECK_KIND => POLICY_IDENTIFIER});

ASSERTION_KIND ::= RM_ASSERTION_KIND | ID_ASSERTION_KIND

CHECK_KIND ::= IDENTIFIER
              Pre'Class
              Post'Class
              Type_Invariant'Class |
              Invariant'Class
```

The identifiers `Name` and `Policy` are not allowed as `CHECK_KIND` values. This avoids confusion between the two possible syntax forms for this pragma.

```
POLICY_IDENTIFIER ::= ON | OFF | CHECK | DISABLE | IGNORE
```

This pragma is used to set the checking policy for assertions (specified by aspects or pragmas), the `Debug` pragma, or additional checks to be checked using the `Check` pragma. It may appear either as a configuration pragma, or within a declarative part of package. In the latter case, it applies from the point where it appears to the end of the declarative region (like pragma `Suppress`).

The `Check_Policy` pragma is similar to the predefined `Assertion_Policy` pragma, and if the check kind corresponds to one of the assertion kinds that are allowed by `Assertion_Policy`, then the effect is identical.

2.30 Pragma Compile_Time_Error

Syntax:

```
pragma Compile_Time_Error
    (boolean_EXPRESSION, static_string_EXPRESSION);
```

This pragma can be used to generate additional compile time error messages. It is particularly useful in generics, where errors can be issued for specific problematic instantiations. The first parameter is a boolean expression. The pragma ensures that the value of an expression is known at compile time, and has the value False. The set of expressions whose values are known at compile time includes all static boolean expressions, and also other values which the compiler can determine at compile time (e.g., the size of a record type set by an explicit size representation clause, or the value of a variable which was initialized to a constant and is known not to have been modified). If these conditions are not met, an error message is generated using the value given as the second argument. This string value may contain embedded ASCII.LF characters to break the message into multiple lines.

2.31 Pragma Compile_Time_Warning

Syntax:

```
pragma Compile_Time_Warning
    (boolean_EXPRESSION, static_string_EXPRESSION);
```

Same as pragma Compile_Time_Error, except a warning is issued instead of an error message. If switch ‘-gnatw-C’ is used, a warning is only issued if the value of the expression is known to be True at compile time, not when the value of the expression is not known at compile time. Note that if this pragma is used in a package that is with’ed by a client, the client will get the warning even though it is issued by a with’ed package (normally warnings in with’ed units are suppressed, but this is a special exception to that rule).

One typical use is within a generic where compile time known characteristics of formal parameters are tested, and warnings given appropriately. Another use with a first parameter of True is to warn a client about use of a package, for example that it is not fully implemented.

In previous versions of the compiler, combining ‘-gnatwe’ with Compile_Time_Warning resulted in a fatal error. Now the compiler always emits a warning. You can use [Pragma Compile_Time_Error], page 18, to force the generation of an error.

2.32 Pragma Complete_Representation

Syntax:

```
pragma Complete_Representation;
```

This pragma must appear immediately within a record representation clause. Typical placements are before the first component clause or after the last component clause. The effect is to give an error message if any component is missing a component clause. This pragma may be used to ensure that a record representation clause is complete, and that this invariant is maintained if fields are added to the record in the future.

2.33 Pragma Complex_Representation

Syntax:

```
pragma Complex_Representation
    ([Entity =>] LOCAL_NAME);
```

The **Entity** argument must be the name of a record type which has two fields of the same floating-point type. The effect of this pragma is to force gcc to use the special internal complex representation form for this record, which may be more efficient. Note that this may result in the code for this type not conforming to standard ABI (application binary interface) requirements for the handling of record types. For example, in some environments, there is a requirement for passing records by pointer, and the use of this pragma may result in passing this type in floating-point registers.

2.34 Pragma Component_Alignment

Syntax:

```
pragma Component_Alignment (
    [Form =>] ALIGNMENT_CHOICE
    [, [Name =>] type_LOCAL_NAME]);

ALIGNMENT_CHOICE ::=
    Component_Size
  | Component_Size_4
  | Storage_Unit
  | Default
```

Specifies the alignment of components in array or record types. The meaning of the **Form** argument is as follows:

‘Component_Size’

Aligns scalar components and subcomponents of the array or record type on boundaries appropriate to their inherent size (naturally aligned). For example, 1-byte components are aligned on byte boundaries, 2-byte integer components are aligned on 2-byte boundaries, 4-byte integer components are aligned on 4-byte boundaries and so on. These alignment rules correspond to the normal rules for C compilers on all machines except the VAX.

‘Component_Size_4’

Naturally aligns components with a size of four or fewer bytes. Components that are larger than 4 bytes are placed on the next 4-byte boundary.

‘Storage_Unit’

Specifies that array or record components are byte aligned, i.e., aligned on boundaries determined by the value of the constant **System.Storage_Unit**.

‘Default’

Specifies that array or record components are aligned on default boundaries, appropriate to the underlying hardware or operating system or both. The **Default** choice is the same as **Component_Size** (natural alignment).


```

    for L2'Scalar_Storage_Order use Low_Order_First;

    type L2a is new L2;

    package Inner is
        type H3 is record
            a : Integer;
        end record;

        pragma Default_Scalar_Storage_Order (Low_Order_First);

        type L4 is record
            a : Integer;
        end record;
    end Inner;

    type H4a is new Inner.L4;

    type H5 is record
        a : Integer;
    end record;
end DSS01;

```

In this example record types with names starting with ‘L’ have *Low_Order_First* scalar storage order, and record types with names starting with ‘H’ have *High_Order_First*. Note that in the case of H4a, the order is not inherited from the parent type. Only an explicitly set *Scalar_Storage_Order* gets inherited on type derivation.

If this pragma is used as a configuration pragma which appears within a configuration pragma file (as opposed to appearing explicitly at the start of a single unit), then the binder will require that all units in a partition be compiled in a similar manner, other than run-time units, which are not affected by this pragma. Note that the use of this form is discouraged because it may significantly degrade the run-time performance of the software, instead the default scalar storage order ought to be changed only on a local basis.

2.48 Pragma Default_Storage_Pool

Syntax:

```
pragma Default_Storage_Pool (storage_pool_NAME | null);
```

This pragma is standard in Ada 2012, but is available in all earlier versions of Ada as an implementation-defined pragma. See Ada 2012 Reference Manual for details.

2.49 Pragma Depends

Syntax:

```
pragma Depends (DEPENDENCY_RELATION);
```

```
DEPENDENCY_RELATION ::=
```

```

    null
  | (DEPENDENCY_CLAUSE {, DEPENDENCY_CLAUSE})

DEPENDENCY_CLAUSE ::=
    OUTPUT_LIST =>[+] INPUT_LIST
  | NULL_DEPENDENCY_CLAUSE

NULL_DEPENDENCY_CLAUSE ::= null => INPUT_LIST

OUTPUT_LIST ::= OUTPUT | (OUTPUT {, OUTPUT})

INPUT_LIST ::= null | INPUT | (INPUT {, INPUT})

OUTPUT ::= NAME | FUNCTION_RESULT
INPUT  ::= NAME

```

where FUNCTION_RESULT is a function Result attribute_reference

For the semantics of this pragma, see the entry for aspect `Depends` in the SPARK 2014 Reference Manual, section 6.1.5.

2.50 Pragma Detect_Blocking

Syntax:

```
pragma Detect_Blocking;
```

This is a standard pragma in Ada 2005, that is available in all earlier versions of Ada as an implementation-defined pragma.

This is a configuration pragma that forces the detection of potentially blocking operations within a protected operation, and to raise `Program_Error` if that happens.

2.51 Pragma Disable_Atomic_Synchronization

Syntax:

```
pragma Disable_Atomic_Synchronization [(Entity)];
```

```
pragma Enable_Atomic_Synchronization [(Entity)];
```

Ada requires that accesses (reads or writes) of an atomic variable be regarded as synchronization points in the case of multiple tasks. Particularly in the case of multi-processors this may require special handling, e.g. the generation of memory barriers. This synchronization is performed by default, but can be turned off using pragma `Disable_Atomic_Synchronization`. The `Enable_Atomic_Synchronization` pragma turns it back on.

The placement and scope rules for these pragmas are the same as those for `pragma Suppress`. In particular they can be used as configuration pragmas, or in a declaration sequence where they apply until the end of the scope. If an `Entity` argument is present, the action applies only to that entity.

- * `FILE_NAME` is the short name (with no directory information) of the Ada source file for `U`, using the required syntax for the underlying file system (e.g. case is significant if the underlying operating system is case sensitive). If `U` is a package and `E` is a subprogram declared in the package specification and its full declaration appears in the package body, then the relevant source file is the one for the package specification; analogously if `U` is a generic package.
- * If `E` is not declared in a generic instantiation (this includes generic subprogram instances), the source trace includes only one source line reference. `LINE_NUMBER` gives the line number of the occurrence of the declaration of `E` within the source file (as a decimal literal without an exponent or point).
- * If `E` is declared by a generic instantiation, its source trace (from left to right) starts with the source location of the declaration of `E` in the generic unit and ends with the source location of the instantiation, given in square brackets. This approach is applied recursively with nested instantiations: the rightmost (nested most deeply in square brackets) element of the source trace is the location of the outermost instantiation, and the leftmost element (that is, outside of any square brackets) is the location of the declaration of `E` in the generic unit.

Examples:

```
pragma Eliminate (Pkg0, Proc);
-- Eliminate (all overloadings of) Proc in Pkg0

pragma Eliminate (Pkg1, Proc,
                  Source_Location => "pkg1.ads:8");
-- Eliminate overloading of Proc at line 8 in pkg1.ads

-- Assume the following file contents:
--   gen_pkg.ads
--   1: generic
--   2:   type T is private;
--   3: package Gen_Pkg is
--   4:   procedure Proc(N : T);
--   ...   ...
--   ... end Gen_Pkg;
--
--   q.adb
--   1: with Gen_Pkg;
--   2: procedure Q is
--   3:   package Inst_Pkg is new Gen_Pkg(Integer);
--   ...   -- No calls on Inst_Pkg.Proc
--   ... end Q;

-- The following pragma eliminates Inst_Pkg.Proc from Q
pragma Eliminate (Q, Proc,
                  Source_Location => "gen_pkg.ads:4[q.adb:3]");
```


