Static Analysis Via Compiler

Overview

Typographical Styles Used

- *This* is a definition
- `this/is/a.path`
- `code is highlighted`
- `commands are emphasised --like-this`

Introduction

- GNAT can be configured to perform static analysis
  - Warnings enabled via compiler switches
- GNAT can be told that a subset of the language will be adhered to by the source code
  - Via language-defined `pragma Restrictions`
  - Affects code generation and run-time library candidates
  - Useful for certification
- GNAT's analysis is extensive, but not without limitations
  - A compiler rather than a static analyzer
  - CodePeer will be used as a counter-example

GNAT Warnings

Warning Categories

- Definite errors
- Probable errors
- Possible mismatches with user expectations
- Redundant code
- Representation-related warnings
  - Biased integer representation, etc.
- See *GNAT User's Guide* for all switches and their meanings

Controlling Warnings With Switches

- Activated with option `-gnatw[x]`
  - Where *x* is a character(s) specific to a warning
- Deactivated with capitalized version of switch
  - E.g., `-gnatwc` activates, `-gnatwC` deactivates
- GCC back-end offers distinct warnings too
- Warnings for nasty cases are enabled by default
  - Unintentional address clause overlays
  - Others...

Warnings Example

```
1  function Bad (B1, B2 : Boolean) return Integer is
2     Result : Integer;
3  begin
4     Result := Result + 1;
5     if B1 then
6        return Result;
7     end if;
8     Result := Bad (B1, B2);
9  end Bad;
```

```
gcc -c -gnatwa bad.adb
```

```
bad.adb:4:14: warning: "Result" may be referenced before it has a value [enabled by default]
bad.adb:8:04: warning: possibly useless assignment to "Result", value might not be referenced [-gnatwm]
bad.adb:8:11: warning: "return" statement missing following this statement [enabled by default]
bad.adb:8:11: warning: Program_Error will be raised at run time [enabled by default]
```

Definite Errors

- Compiler detects a runtime failure
  - Compiler can tell that an assertion is always false
  - Exceptions raised but not caught locally and `No_Exception_Propagation` restriction is applied

Definite Error Examples

```
1  pragma Restrictions (No_Exception_Propagation);
2  procedure Test (Failure : Boolean) is
3  begin
4     if Failure then
5      raise Constraint_Error;
6   end if;
7  end Test;
```

```
test.adb:5:07: warning: pragma Restrictions (No_Exception_Propagation) in effect [-gnatw.x]
test.adb:5:07: warning: execution may raise unhandled exception [-gnatw.x]
```

```
1  procedure Test (Param : in out Integer) is
2  begin
3     pragma Assert (Integer'object_size = 64);
4     Param := Param + 1;
5  end Test;
```

```
test.adb:3:19: warning: assertion would fail at run time [-gnatw.a]
```

Probable Errors

- Errors where compiler thinks coder made a mistake
  - Conditions that are always false or always true
  - Unused formal parameters
    - Can apply `pragma Unreferenced`, especially in OOP case
  - Variables that could be declared as constants
    - Not so much an error but should be heeded
  - Variables assigned but not read
  - Variables read but not assigned
  - Unchecked conversions with different source and target type sizes
  - Unlikely modulus value in type declaration
  - Suspicious actual parameter ordering
  - Missing parentheses may be confusing

Probable Errors - Source Code

```
1  with Unchecked_Conversion;
2  package body Examples is
3
4     function Convert is new Unchecked_Conversion (Integer, Character);
5     type Mod_T is mod 2 * 32;
6
7     procedure Example (A, B, C :     Natural;
8                        D       : out Natural) is
9        E : Natural := A * B;
10       F : Natural;
11    begin
12       if E >= 0 then
13          D := D + A / B;
14          F := E;
15       end if;
16    end Example;
17
18    procedure Test (A, B, C :     Integer;
19                    D       : out Integer) is
20    begin
21       Example (A, C, B, D);
22       D := -D mod B;
23    end Test;
24
25 end Examples;
```

Probable Errors - Results

```
examples.adb:3:04: warning: types for unchecked conversion have different sizes [-gnatwz]
examples.adb:4:24: warning: suspicious "mod" value, was ** intended? [-gnatw.m]
examples.adb:6:13: warning: formal parameter "C" is not referenced [-gnatwu]
examples.adb:8:07: warning: "E" is not modified, could be declared constant [-gnatwk]
examples.adb:9:07: warning: variable "F" is assigned but never read [-gnatwm]
examples.adb:11:12: warning: condition can only be False if invalid values present [-gnatwc]
examples.adb:11:12: warning: condition is always True [-gnatwc]
examples.adb:13:15: warning: "D" may be referenced before it has a value [enabled by default]
examples.adb:21:07: warning: actuals for this call may be in wrong order [-gnatw.p]
examples.adb:22:12: warning: unary minus expression should be parenthesized here [enabled by default]
```

Probable Errors - Explanations

- Line 5 - Coder probably meant `2 ** 32`
  - But maybe not? It could be a bit location
- Line 12 - `E` is `natural`, so it can never be less than zero (without invalid data)
- Line 13 - `D` is an `out` parameter, so there is no guarantee on it's initial value
- Line 22 - Did you mean `-(D mod B)` or `(-D) mod B`?

Redundant Code

- Comparing boolean expression to boolean value
- Type conversion when the entity is already of the target type

Redundant Code - Examples

```ada
package body Redundant_Code is

   procedure Test
     (A, B, C :        Integer;
      D       : in out Integer) is
   begin
      if (A > B) = True then
         D := D - 1;
      end if;
      D := D - Integer (C);
   end Test;

end Redundant_Code;
```

```
redundant_code.adb:7:18: warning: comparison with True is redundant [-gnatwr]
redundant_code.adb:10:16: warning: redundant conversion, "C" is of type "Integer" [-gnatwr]
```

Controlling Warnings With A Single Switch

- Switch `-gnatwa` enables almost all warnings
  - Those typically useful
  - Good balance between actual problems and false positives
- Switch `-gnatw.e` enables absolutely all warnings
  - Including those not activated by `-gnatwa`
  - Not recommended for typical use
  - Likely generates many warnings you'll end up ignoring
  - But you might want some of them, individually

Highly Optional Warnings `-gnatw.e`

- Implicit dereferencing (missing optional `.all`)
- Activate tagging (warning messages tagged with certain strings)
- Suspicious Subp'Access
- Warnings for GNAT sources
- Hiding (Potentially confusing hiding of declarations)
- Holes/gaps in records
- Redefinition of names in package `Standard`
- Elaboration pragmas
- List inherited aspects
- Atomic synchronization
- Modified but unreferenced parameters
- Out of order record representation clauses
- Overridden size clauses
- Tracking of deleted conditional code
- Unordered enumeration types
- Warnings Off pragmss (flags unnecessary pragmas)
- Activate information messages for why package needs a body

Unordered Enumeration Value Comparisons

- Most enumerations are not semantically ordered

```
-- not semantically ordered
type Colors_T is (Red, Yellow, Green);
-- semantically ordered
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

- Comparisons other than equality are suspect

```
14 if Current_Color > Yellow then -- must be Green, so go
```

- Maintainers (you!) may change order later

```
type Colors_T is (Green, Yellow, Red);
```

- GNAT **pragma** `Ordered` can be used say that such comparisons make sense

```
pragma Ordered (Days);
```

- Can set warning `-gnatw.u` to flag unordered relations

```
examples.adb:14:32: warning: comparison on unordered enumeration type "Colors_t" declared at colors.ads:4 [-gnatw.u]
```

Notifications of Deleted Conditional Code

- Also known as deactivated code
- Applies to if-statements and case-statements
- May be useful in certified applications

```
3  procedure Test (A : in out Integer) is
4  begin
5     if False then
6        Put_Line ("Commented out for now");
7     else
8        Put_Line (A'image);
9     end if;
10 end Test;
```

```
examples.adb:6:10: warning: this code can never be executed and has been deleted [-gnatwt]
```

Controlling Warnings Within the Source Text

- Via **pragma** Warnings
  - See **Implementation Defined Pragmas** in *GNAT Reference Manual*
- Syntax
  - All have an optional string literal parameter Reason ignored by compiler but perhaps processed by other tools

```
pragma Warnings ([TOOL_NAME,] DETAILS [, REASON]);
```

```
DETAILS ::= On | Off
```

- Enable/Disable all warnings

```
DETAILS ::= On | Off, Local_Name
```

- Enable/Disable all warnings for Local_Name

```
DETAILS ::= Static_String_Expression
```

- Enable/Disable warnings based on compiler switches specified in Static_String_Expression

```
DETAILS ::= On | Off, Static_String_Expression
```

- Enable/Disable all warnings based on warning message specified in Static_String_Expression

```
TOOL_NAME ::= SPARK | GNATprove
```

- Control which tool responds to pragma

```
REASON ::= Reason => STRING_LITERAL {& STRING_LITERAL}
```

- Informational message that can be parsed by external tools

Pragma Warnings Usage Examples

- All warnings off in this region of code only

```
pragma Warnings (Off);
Free (X);
pragma Warnings (On);
```

- All warnings off for this object, throughout its scope

```
New_Tgt_Node : Counter;
pragma Warnings (Off, New_Tgt_Node);
```

- All warnings off that emit messages matching this text, in this region of code only

```
-- Optional; matches any message text
pragma Warnings (Off, "loop range is null*");
--  On monoprocessor targets, the following loop will
--  never execute (no other CPUs).
for CPU_Id in CPU'First + 1 .. CPU'Last loop
   Start_CPU (CPU_Id);
end loop ;
pragma Warnings (On, "loop range is null*");
```

GNAT Style Checking

"Style" Checking

- Style rules we use within AdaCore
    - Not a general coding standards checker (see GNATCHECK)
    - Some are arbitrary
    - Main thing is to be consistent
- Categories of checks
    - Layout/presentation
    - Sound Engineering
- Note that you don't have to use any/all of these!

GNAT Style Enforcement Switches

- Activated with option `-gnatyxx`
  - Where **xx** is replaced with list of style check parameters
- Deactivated after minus (-):
  - `-gnatyc` activates, `-gnaty-c` deactivates
- `-gnaty` activates most style warnings (also `-gnatyY`)
  - Equivalent to `-gnaty3abcefhiklmnprst`
  - (Descriptions on following pages)
- `-gnatyN` suppresses all style warnings
- See *GNAT User's Guide* section 3.2.5 for all the options available

GNAT Modes

- Internal GNAT implementation mode `-gnatg` → `-gnatyg -gnatw.ge`
- GNAT-Style mode `-gnatyg` → `-gnatyydISuxz`
  - `y` All standard check options
  - `d` No DOS line-terminators
  - `I` No **explicit** `in` keyword
  - `S` `then` / `else` statements on **different** line
  - `u` No unnecessary blank lines
  - `x` No extra parentheses in conditionals
  - `z` No extra parentheses in operations
- GNAT source warnings `-gnatw.g` (next slide)
- Activate every optional warning `-gnatw.e`

GNAT Source Warnings `-gnatw.g`

- *GNAT Source warnings* meaning may evolve and switches may change
- As of now, `-gnatw.g` → `-gnatwAao.q.s.CI.V.X.Z`
  - `Aao` Reset warnings to `-gnatwa`
  - `.q` Questionable / ineficient layout of record type
  - `.s` Overriden size clause (sizes mismatch)
  - `.C` No warning for incomplete component representation clause
  - `I` No warning on **`with`** of internal GNAT package
  - `.V` No info message on non-default bit-order
  - `.X` No warning for `Restriction (No_Exception_Propagation)`
  - `.Z` No warning for `'Size mod 'Alignment /= 0`

Layout and Presentation Checks

Style check	Behavior
1-9	check indentation
a	check attribute casing
b	check no blanks at end of lines
c	check comment format (two spaces)
C	check comment format (one space)
d	check no DOS line terminators
f	check no form feeds/vertical tabs in source
h	check no horizontal tabs in source
i	check if-then layout
k	check casing rules for keywords
l	check reference manual layout
m	check line length <= 79 characters
Mnn	check line length <= nn characters
n	check casing of package Standard identifiers
o	check subprogram bodies in alphabetical order
p	check pragma casing
r	check casing for identifier references
S	check separate lines after THEN or ELSE
t	check token separation rules
u	check no unnecessary blank lines

Layout and Presentation Example

```
79  -- Procedure to find the defining name for the node
80  procedure Find_Defining_Name (Node : Lal.Ada_Node'Class) is
81     Parent : Lal.Ada_Node := node.Parent;
82  begin
83     --  Go up the tree until we find what we are looking for
84     Search_Loop:
85     While not Parent.Is_Null loop
86        exit Search_Loop when Names.Map_Size = Natural'last;
87        if Parent.Kind = Lalco.Ada_Defining_Name then
88           if Valid_Length (Qualified_Name) then
89             Names.Add_Name (Qualified_Name);
90           end if;
91        end if;
92        Parent := Parent.Parent;
93     end loop Search_Loop;
94  end Find_Defining_Name;
```

Message	Caused by
obfuscate.adb:79:07: (style) space required	*-gnatyc*
obfuscate.adb:81:32: (style) bad casing of "Node" declared at line 80	*-gnatyr*
obfuscate.adb:84:18: (style) space required	*-gnatyt*
obfuscate.adb:85:07: (style) reserved words must be all lower case	*-gnatyk*
obfuscate.adb:86:57: (style) bad capitalization, mixed case required	*-gnatya*
obfuscate.adb:89:15: (style) bad indentation	*-gnaty3*

Sound Engineering Checks

Style check	Behavior
A	check array attribute indexes
B	check no use of AND/OR for boolean expressions
e	check end/exit labels present
I	check mode in
Lnn	check max nest level < nn
O	check overriding indicators
s	check separate subprogram specs present
x	check extra parentheses around conditionals

Sound Engineering Example

```
4  package Example is
5     Count : Natural;
6     type Tagged_T is tagged null record;
7     procedure Primitive (R : in Tagged_T);
8     type Child_T is new Tagged_T with record
9        Field : Natural;
10    end record;
11    procedure Primitive (R : in Child_T);
12 end Example;
13
14 package body Example is
15    procedure Primitive (R : in Tagged_T) is
16    begin
17       if (Count > 0) then Count := 0; end if;
18    end Primitive;
19    procedure Primitive (R : in Child_T) is
20    begin
21       Lup :
22       while (Count > 0) and (Count < 100) loop
23          Count := Count + R.Field;
24          exit when Count = 50;
25       end loop Lup;
26    end Primitive;
27 end Example;
```

Message	Caused by
examples.adb:7:32: (style) "in" should be omitted	*-gnatyI*
examples.adb:11:07: (style) missing "overriding" indicator in declaration of "Primitive"	*-gnatyO*
examples.adb:17:13: (style) redundant parentheses	*-gnatyx*
examples.adb:17:30: (style) no statements may follow "then" on same line	*-gnatyS*
examples.adb:19:07: (style) missing "overriding" indicator in body of "Primitive"	*-gnatyO*
examples.adb:22:28: (style) "and then" required	*-gnatyB*
examples.adb:24:13: (style) "exit Lup" required	*-gnatye*

Warnings Versus Errors

- If you must ensure issues are caught, failing to compile is the most rigorous enforcement
- Compiler can be told to treat warnings as errors
  - Thus code rejected at compile-time
- Use switch `-gnatwe`
  - Warnings become errors
  - Style violations become errors too
  - Warning messages still appear but no code generation

IDE Integration (Project Properties Editor)

Properties for Amazing

General
Sources
Dependencies
Languages
Directories
Files
Main
Naming
Ada
Build
Toolchain
Make
Directories
Switches
CodePeer
GNATstack
Pretty Printer
GNATprove
GNATcheck
Builder
Ada
Binder
Ada Linker
CodePeer

Code generation
Full optimization
Inlining
Unroll loops
Link time optimization
Position independent code
Always generate ALI file
Separate function sections
Separate data sections

Messages
Full errors
Warnings:
Validity checking mode:
Style checks:

Version
Ada version
Default Ada 83 Ada 95 Ada 2005 Ada 2012 Ada 2020 GNAT Extensions

Stack usage
Generate stack usage information

-O2 -gnat12 -gnatwa

Run-time checks
Overflow checking
Suppress all checks
Stack checking
Dynamic elaboration

Debugging
Debug Information
Enable assertions

Apply changes to:
Show as hierarchy
Project
Amazing

Scenario
MODE
debug
release
Console
Win32
ANSI

Save Cancel

Warnings Dialog

Warnings

Global switches

- Activate most optional warnings (-gnatwa)
- Treat warnings and style checks as errors (-gnatwe)
- Activate every optional warning (-gnatw.e)
- Suppress all warnings

Warnings

- Failing assertions
- Biased representation
- Conditionals
- Missing component clauses
- Unknown condition in Compile_Time_Warning
- Implicit dereferencing
- Activate tagging of warning and info messages
- Unreferenced formals
- Suspicious Subp'Access
- Unrecognized pragmas
- Warnings used for GNAT sources
- Hiding
- Holes/gaps in records
- Implementation units
- Overlapping actuals
- Obsolescent features (Annex J)
- Late declarations of tagged type primitives
- Variables that could be constants
- Redefinition of names in standard
- Elaboration pragmas
- List inherited aspects
- Modified but unreferenced variables
- Suspicious modulus values
- Atomic synchronization
- Address clause overlays
- Modified but unreferenced out parameters
- Ineffective pragma Inlines
- Parameter ordering
- Questionable missing parentheses
- Questionable layout of record types
- Redundant constructs
- Object renaming function
- Out-of-order record representation clauses
- Overridden size clauses
- Tracking of deleted conditional code
- Suspicious contracts
- Unused entities
- Unordered enumeration types
- Unassigned variables
- Activate info messages for non-default bit order
- Wrong low bound assumption
- Warnings Off pragmas
- Export/Import pragmas
- No_Exception_Propagation mode
- Ada compatibility issues
- Activate information messages for why package spec needs body
- Unchecked conversions
- Size not a multiple of alignment

OK

Style Checks Dialog

Style checks

Check indentation: 0

- Check attribute casing
- Use of array index numbers in array attributes
- Blanks not allowed at statement end
- Check Boolean operators
- Check comments, double space
- Check comments, single space
- Check no DOS line terminators present
- Check end/exit labels
- No form feeds or vertical tabs
- No horizontal tabs
- Check if-then layout
- Check mode IN keywords
- Check keyword casing
- Check layout

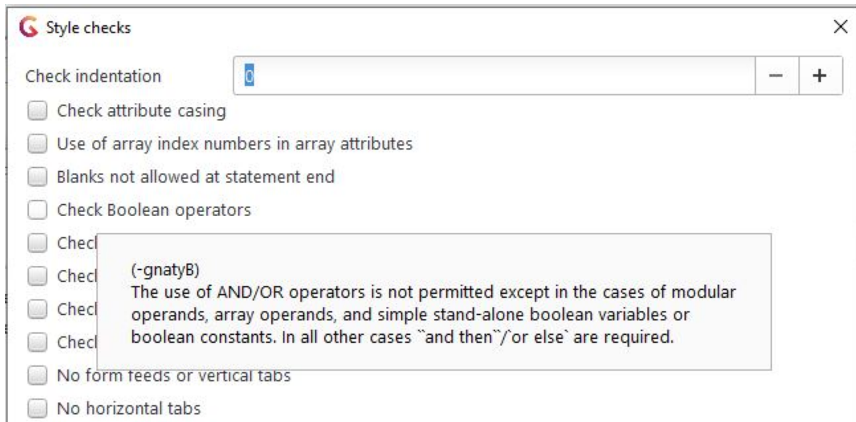
Set maximum nesting level: 0

Set maximum line length: 0

- Check casing of entities in Standard
- Check order of subprogram bodies
- Check that overriding subprograms are explicitly marked as such
- Check pragma casing
- Check references
- Check separate specs
- Check no statements after then/else
- Check token spacing
- Check unnecessary blank lines
- Check extra parentheses

OK

Dialog Pop-Ups Explain Style Options



Language Subset Definitions

Definition of Language Subsets

- Uses language-defined **pragma** Restrictions

```
pragma Restrictions (restriction{, restriction});
restriction ::= restriction_identifier |
                restriction_parameter_identifier =>
                     restriction_parameter_argument
```

- Provides control over many features
  - Tasking, exceptions, dispatching, code generation, elaboration, etc.
- Benefits
  - Faster execution on compatible run-time library
  - Safer coding
  - Certification restrictions compliance
  - Compiler/target portability
- Restrictions can also be added by setting up a runtime profile via `Pragma Profile(<runtime>)` which enables all restrictions implemented in the specified runtime

Example Restriction & Violation Message

```
1    pragma Restrictions (No_Implicit_Heap_Allocations);
2
3    with Ada.Command_Line;
4    package Lib_Level is
5       -- Command_Name returns an unconstrained type
6       Command_Name : constant String := Ada.Command_Line.Command_Name;
7    end Lib_Level;
```

```
lib_level.ads:6:04: error: violation of restriction "No_Implicit_Heap_Allocations" at line 1
```

Only happens for library level package specs, not just any package and not package bodies.

Restriction Identifiers

- All language-defined identifiers are implemented
  - Core restrictions (see 13.12.1)
  - Real-time tasking restrictions (see D.7)
  - High integrity restrictions (see H.4)
- GNAT defines additional restriction identifiers
- All restrictions, both language-defined and GNAT-defined, are listed and described in the *GNAT Reference Manual*

Restriction Categories

- Portability
- Allocation
- Access Types & Values
- Exceptions
- OOP
- Tasking
- Real-Time Programming
- Code Generation
- Miscellaneous
- GNAT defines additional restrictions in all these categories
  - We examine some of them here...

Applying Restriction Identifiers

- In source or in configuration file
  - Configuration file name should be specified in the GPR file

```
package Compiler is
  for Local_Configuration_Pragmas
      use "configuration_pragmas.adc";
end Compiler;
```

  - Or, if not GPR file is in use, in the default config file `gnat.adc`

```
pragma Restrictions (No_Implicit_Heap_Allocations);
pragma Restrictions (No_Implicit_Conditionals);
pragma Restrictions (No_Entry_Calls_In_Elaboration_Code);
```

- GNATBIND can list all restrictions that could be applied to the code corresponding to a given ALI file
  - Via `-r` switch
  - Useful for code audit, and code generation control

OOP Restrictions

- `No_Dispatch` (RM H.4)
  - Ensures no occurrences of `T'Class` for any tagged type `T`
  - Prevents dynamic dispatching (but also other usage)
- `No_Dispatching_Calls` (GNAT)
  - Ensures generated code involves no dispatching calls
  - Allows
    - Record extensions
    - Classwide membership tests
    - Other classwide features
  - Does not allow involving implicit dispatching
  - Comparable to `No_Dispatch`
    - Except allows all classwide constructs that do not imply dispatching

Quiz

```
   package Definition is
      type T is tagged record
         Data : Natural;
      end record;
      procedure P (X : T);
      type Dt is new T with record
         More_Data : Natural;
      end record;
      not overriding procedure Q (X : Dt);
   end Definition;
1  pragma Restrictions (No_Dispatching_Calls);
2
3  with Definition; use Definition;
4  procedure Demo (O : T'class) is
5     N : Natural := O'size;
6     C : T'class := O;
7  begin
8     if O in Dt'class then
9        Q (Dt (O));
10    else
11       P (O);
12    end if;
13 end Demo;
```

Which line(s) violate the restriction?

A. 5, 6, 8, 9, 11
B. 11
C. 5, 6, 11
D. No violations

Quiz

```
    package Definition is
       type T is tagged record
          Data : Natural;
       end record;
       procedure P (X : T);
       type Dt is new T with record
          More_Data : Natural;
       end record;
       not overriding procedure Q (X : Dt);
    end Definition;
1   pragma Restrictions (No_Dispatching_Calls);
2
3   with Definition; use Definition;
4   procedure Demo (O : T'class) is
5      N : Natural := O'size;
6      C : T'class := O;
7   begin
8      if O in Dt'class then
9         Q (Dt (O));
10     else
11        P (O);
12     end if;
13  end Demo;
```

Which line(s) violate the restriction?

A. 5, 6, 8, 9, 11
B. 11
C. ***5, 6, 11***
D. No violations

- Line 5 - Dispatch needed to determine size of O
- Line 6 - Just a memory copy (no dispatching)
- Line 8 - Membership not a dispatching call
- Line 9 - Type conversion so no dispatching
- Line 11 - Dispatch needed to find correct P

Exceptions Restrictions Form A Spectrum

- `No_Exceptions` (RM H.4)
  - No raise statements and no handlers
- `No_Exception_Handlers` (GNAT)
  - No exception handlers
  - Raised exception raised result in call to the *last chance handler*
- `No_Exception_Propagation` (GNAT)
  - Exceptions never propagated out of subprogram
  - Handlers are allowed
    - May not contain an exception occurrence identifier
  - Handler must be in same subprogram
    - Raise is essentially a `goto` statement
  - Any other raise statement considered unhandled

No_Implicit_Conditionals (GNAT)

- Generated code does not contain any implicit conditionals
  - E.g., comparisons of composite objects (maybe)
  - E.g., the Max/Min attributes (maybe)
- Modifies the generated code where possible, or rejects any construct that would otherwise generate an implicit conditional
- If rejected, the programmer must make the condition explicit in the source

No_Implicit_Loops (GNAT)

- Ensures generated code does not contain any implicit loops
  - Actual code

    ```
    X : array (1 .. 100) of Integer := (1, 2, others => 3);
    ```
  - Generated code

    ```
    x (1) := 1;
    x (2) := 2;
    k : integer := 2;
    while k  <  100 loop
       k := k + 1;
       x (k) := 3;
    end loop;
    ```
- Modifies code generation approach where possible, or rejects construct
- If rejected, programmer must make loop explicit
- Can improve code performance

GNAT Initialization Restrictions

- `No_Initialize_Scalars`
  - No unit in partition compiled with `pragma Initialize_Scalars`
  - Allows generation of more efficient code
- `No_Default_Initialization`
  - Forbids any default variable initialization of any kind

```
1  pragma Restrictions (No_Default_Initialization);
2  procedure Demo is
3     type Record_T is record
4        Field : Integer := 42;
5     end record;
6     Bad  : Record_T;
7     Good : Record_T := (Field => 42);
```

```
demo.adb:6:04: error: violation of restriction "No_Default_Initialization" at line 1
```

Miscellaneous GNAT Restrictions

- `No_Direct_Boolean_Operators`
  - Short-circuit forms required everywhere
  - More restrictive than GNAT style switch
- `No_Elaboration_Code`
  - No elaboration code is generated
  - Not the same as `pragma Preelaborate`
- `No_Enumeration_Maps`
  - No `'Image` and `'Value` applied to enumeration types
    - No need to keep strings
  - Compare to `pragma Discard_Names`
    - Applies to enumeration types, tagged types, and exceptions

GNAT Stream Restrictions

- `No_Stream_Optimizations`
  - Performs all I/O operations on a per-character basis
    - Rather than larger whole-array object basis
- `No_Streams`
  - No stream objects created and no use of stream attributes
  - Less code generated
  - Worth considering if using tagged types on memory-constrained targets

No_Finalization (GNAT)

- Disables features described in *Ada Reference Manual* section 7.6 plus all forms of code generation supporting them
  - Initialization as well as finalization
- Following types are no longer controlled types
  - `Ada.Finalization.Controlled` and `Limited_Controlled`
  - Types derived from `Controlled` or `Limited_Controlled`
  - Class-wide types
  - Protected types
  - Task types
  - Array and record types with controlled components
- Compiler no longer generates code to initialize, finalize or adjust objects

Getting Representation Info

Traceability from Source Code to Object Code

- Expanded sources can be viewed
  - Shows how tasks implemented, aggregates expanded, etc.
  - Facilitates certification activities
- Expanded code syntax described in *GNAT User's Guide*
- Enabled via `-gnatG`
  - Add `-gnatL` to intersperse source lines as comments

Expanded Code Example

- Actual code

```
1 procedure Demo is
2   X : array (1 .. 100) of Integer := (1, 2, others => 3);
3 begin
4   null;
5 end Demo;
```

- Generated code

```
-- 1: procedure Demo is
procedure demo is
-- 2:    X : array (1 .. 100) of Integer := (1, 2, others => 3);
   [type demo__TxB is array (1 .. 100 range <>) of integer]
   freeze demo__TxB []
   [subtype demo__TxT1b is demo__TxB (1 .. 100)]
   freeze demo__TxT1b []
   x : array (1 .. 100) of integer;
   x (1) := 1;
   x (2) := 2;
   J6b : integer := 2;
   L7b : while J6b < 100 loop
      [constraint_error when
        J6b = 16#7FFF_FFFF#
        "overflow check failed"]
      J6b := integer'succ(J6b);
      x (J6b) := 3;
   end loop L7b;
-- 3: begin
begin
-- 4:    null;
   null;
-- 5: end Demo;
   return;
end demo;
```

See How Types and Objects Are Represented

- Compiler switch shows all representation aspects
  - Size in memory
  - Size required for values
  - Alignment
  - Component sizes
- Reflects user specifications
  - Record type representation
  - Array component sizes
  - et cetera
- Reflects compiler defaults
  - When not specified by application code

Settings for Viewing Representations

- **-gnatR0** No information
- **-gnatR1** Size / alignment for array and record types
- **-gnatR2** Size / alignment for all types and objects
- **-gnatR3** Symbolic expressions for variant record info

- If the switch is followed by an 's' the output is to a file with the name `<file>.rep` where *<file>* is the name of the corresponding source file
- Note `-gnatR` is same as -`gnatR1`

Viewing Data Representations Example

- Performing `gcc -c -gnatR3` on:

```
package Some_Types is
   type Temperature is range -275 .. 1_000;
   type Identity is range 1 .. 127;
   type Info is record
      T  : Temperature;
      Id : Identity;
   end record;
end Some_Types;
```

- Generates:

```
for Temperature'Object_Size use 16;
for Temperature'Value_Size use 11;
for Temperature'Alignment use 2;

for Identity'Object_Size use 8;
for Identity'Value_Size use 7;
for Identity'Alignment use 1;

for Info'Object_Size use 32;
for Info'Value_Size use 24;
for Info'Alignment use 2;
for Info use record
   T  at 0 range  0 .. 15;
   Id at 2 range  0 ..  7;
end record;
```

GNAT versus CodePeer

CodePeer

- A static analyzer
  - Provides deep analysis prior to execution and test
- Helps identify vulnerabilities and bugs
  - Better than the compiler
  - Better than a human!
- Is modular and scalable
  - Can be used on an entire project or a single file
  - Can be configured to be more or less strict
- Is flexible
  - Usable with all Ada language variants
  - Usable with other vendors' compilers

Why Not Just Use the Compiler?

- The compiler does generate useful warnings
    - But CodePeer far exceeds the compiler's analyses
- CodePeer
    - Does much more thorough job
    - Finds problems compiler doesn't look for

How Does GNAT Analysis Work?

- Intraprocedural
  - Ignores interactions between caller and called subprograms
- Flow-sensitive but path- and context-insensitive
  - Recognizes order of statements
  - Ignores effects of conditional statements
  - Ignores calling context
- Low-noise
- Very useful, but not complete

Flow Tracing

```ada
function Example (K : Integer) return Integer is
   A, B, C, D : Integer;
begin
   C := A;
   if K > 4 then
      B := 3;
   end if;
   D := B;
   return D;
end Example;
```

- Compiler results:

```
example.adb:2:04: warning: variable "A" is read but never assigned
```

- CodePeer results

```
example.adb:4:9: high: validity check: A is uninitialized here
example.adb:8:9: medium: validity check: B might be uninitialized
```

Value Tracing

```
function Example (K : Integer) return Integer is
   A : Integer;
begin
   A := 4;
   if A > 3 then
      A := A + 1;
   end if;
   if A > 4 then
      A := A + 1;
   end if;
   return A + K;
end Example;
```

- GNAT does only rudimentary value tracing
  - Traces constant values assigned in straight-line code with no conditions

```
example.adb:5:14: warning: condition is always True
```

- CodePeer does full value tracing

```
example.adb:5:09: warning: condition is always True
example.adb:8:9: medium warning: test always true because A = 5
```

"Intra"procedural vs. "Inter"procedural Analysis

```
1 function Example (K : Integer) return Integer is
2    A, B, C : Integer;
3    function Zero return Integer is (0);
4 begin
5    A := 0;
6    B := K / A;
7    C := B / Zero;
8    return C;
9 end Example;
```

- GNAT only analyzes one routine at a time

```
example.adb:6:13: warning: division by zero [enabled by default]
```

- CodePeer does whole-program analysis

```
example.adb:6:11: high: divide by zero fails here
example.adb:7:11: high: divide by zero fails here: requires (zero'Result) /= 0
```

CodePeer's Capabilities Beyond the Compiler's

- Detecting race conditions in tasking code
- Incremental analysis
  - Historical database preserves results of every run
  - Allows user to focus on new problems or compare against baseline
  - Only the changes need be analyzed
- Contract-based Programming support
  - Can generate contracts automatically from the code
  - Can detect incorrect contracts (statically)
  - Can use existing contracts in further analysis
- Others...

Summary

Summary

- Compiler can generate a large number of useful warnings
- Multiple warning categories supported
  - Layout and presentation
  - Sound engineering coding practices
  - Language subset definitions
- See the docs: we did not examine every possibility
- CodePeer can do much better, and much more
  - And analysis is sound
- You can use these facilities directly but you can also apply them via GNATcheck