GNAT Reference Manual

GNAT Reference Manual , Jan 09, 2026

AdaCore

Table of Contents

1 About This Guide 2
1.1 What This Reference Manual Contains 2
1.2 Conventions 3
1.3 Related Information 3

2 Implementation Defined Pragmas 5
2.1 Pragma Abort_Defer 5
2.2 Pragma Abstract_State 5
2.3 Pragma Ada_83 6
2.4 Pragma Ada_95 7
2.5 Pragma Ada_05 7
2.6 Pragma Ada_2005 7
2.7 Pragma Ada_12 8
2.8 Pragma Ada_2012 8
2.9 Pragma Ada_2022 8
2.10 Pragma Aggregate_Individually_Assign 9
2.11 Pragma Allow_Integer_Address 9
2.12 Pragma Always_Terminates 10
2.13 Pragma Annotate 10
2.14 Pragma Assert 10
2.15 Pragma Assert_And_Cut 11
2.16 Pragma Assertion_Level 11
2.17 Pragma Assertion_Policy 12
2.18 Pragma Assume 13
2.19 Pragma Assume_No_Invalid_Values 14
2.20 Pragma Async_Readers 14
2.21 Pragma Async_Writers 14
2.22 Pragma Attribute_Definition 15
2.23 Pragma C_Pass_By_Copy 15
2.24 Pragma Check 15
2.25 Pragma Check_Float_Overflow 16
2.26 Pragma Check_Name 16
2.27 Pragma Check_Policy 17
2.28 Pragma Comment 18
2.29 Pragma Common_Object 18
2.30 Pragma Compile_Time_Error 19
2.31 Pragma Compile_Time_Warning 19
2.32 Pragma Complete_Representation 19
2.33 Pragma Complex_Representation 20
2.34 Pragma Component_Alignment 20
2.35 Pragma Constant_After_Elaboration 21
2.36 Pragma Contract_Cases 21
2.37 Pragma Convention_Identifier 22

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

2.85 Pragma Independent_Components ... 44
2.86 Pragma Initial_Condition ... 44
2.87 Pragma Initialize_Scalars ... 45
2.88 Pragma Initializes ... 46
2.89 Pragma Inline_Always ... 46
2.90 Pragma Inline_Generic ... 47
2.91 Pragma Interface ... 47
2.92 Pragma Interface_Name ... 47
2.93 Pragma Interrupt_Handler ... 47
2.94 Pragma Interrupt_State ... 48
2.95 Pragma Interrupts_System_By_Default ... 49
2.96 Pragma Invariant ... 49
2.97 Pragma Keep_Names ... 49
2.98 Pragma License ... 50
2.99 Pragma Link_With ... 51
2.100 Pragma Linker_Alias ... 51
2.101 Pragma Linker_Constructor ... 52
2.102 Pragma Linker_Destructor ... 52
2.103 Pragma Linker_Section ... 52
2.104 Pragma Lock_Free ... 53
2.105 Pragma Loop_Invariant ... 54
2.106 Pragma Loop_Optimize ... 55
2.107 Pragma Loop_Variant ... 55
2.108 Pragma Machine_Attribute ... 56
2.109 Pragma Main ... 56
2.110 Pragma Main_Storage ... 56
2.111 Pragma Max_Queue_Length ... 57
2.112 Pragma No_Body ... 57
2.113 Pragma No_Caching ... 57
2.114 Pragma No_Component_Reordering ... 57
2.115 Pragma No_Elaboration_Code_All ... 58
2.116 Pragma No_Heap_Finalization ... 58
2.117 Pragma No_Inline ... 58
2.118 Pragma No_Raise ... 59
2.119 Pragma No_Return ... 59
2.120 Pragma No_Strict_Aliasing ... 59
2.121 Pragma No_Tagged_Streams ... 59
2.122 Pragma Normalize_Scalars ... 60
2.123 Pragma Obsolescent ... 61
2.124 Pragma Optimize_Alignment ... 63
2.125 Pragma Ordered ... 64
2.126 Pragma Overflow_Mode ... 65
2.127 Pragma Overriding_Renamings ... 65
2.128 Pragma Part_Of ... 66
2.129 Pragma Partition_Elaboration_Policy ... 66
2.130 Pragma Passive ... 66
2.131 Pragma Persistent_BSS ... 67

2.132 Pragma Post 67
2.133 Pragma Postcondition 67
2.134 Pragma Post_Class 70
2.135 Pragma Pre 70
2.136 Pragma Precondition 70
2.137 Pragma Predicate 71
2.138 Pragma Predicate_Failure 72
2.139 Pragma Preelaborable_Initialization 72
2.140 Pragma Prefix_Exception_Messages 72
2.141 Pragma Pre_Class 72
2.142 Pragma Priority_Specific_Dispatching 73
2.143 Pragma Profile 73
2.144 Pragma Profile_Warnings 76
2.145 Pragma Program_Exit 76
2.146 Pragma Propagate_Exceptions 76
2.147 Pragma Provide_Shift_Operators 76
2.148 Pragma Psect_Object 77
2.149 Pragma Pure_Function 77
2.150 Pragma Rational 78
2.151 Pragma Ravenscar 78
2.152 Pragma Refined_Depends 78
2.153 Pragma Refined_Global 79
2.154 Pragma Refined_Post 79
2.155 Pragma Refined_State 79
2.156 Pragma Relative_Deadline 80
2.157 Pragma Remote_Access_Type 80
2.158 Pragma Rename_Pragma 80
2.159 Pragma Restricted_Run_Time 81
2.160 Pragma Restriction_Warnings 81
2.161 Pragma Reviewable 81
2.162 Pragma Secondary_Stack_Size 82
2.163 Pragma Share_Generic 83
2.164 Pragma Shared 83
2.165 Pragma Short_Circuit_And_Or 83
2.166 Pragma Short_Descriptors 84
2.167 Pragma Side_Effects 84
2.168 Pragma Simple_Storage_Pool_Type 84
2.169 Pragma Source_File_Name 85
2.170 Pragma Source_File_Name_Project 86
2.171 Pragma Source_Reference 87
2.172 Pragma SPARK_Mode 87
2.173 Pragma Static_Elaboration_Desired 88
2.174 Pragma Stream_Convert 88
2.175 Pragma Style_Checks 89
2.176 Pragma Subprogram_Variant 91
2.177 Pragma Subtitle 92
2.178 Pragma Suppress 92

2.179 Pragma Suppress_All . . . 93
2.180 Pragma Suppress_Debug_Info . . . 93
2.181 Pragma Suppress_Exception_Locations . . . 93
2.182 Pragma Suppress_Initialization . . . 94
2.183 Pragma Task_Name . . . 94
2.184 Pragma Task_Storage . . . 95
2.185 Pragma Test_Case . . . 95
2.186 Pragma Thread_Local_Storage . . . 96
2.187 Pragma Time_Slice . . . 96
2.188 Pragma Title . . . 97
2.189 Pragma Type_Invariant . . . 97
2.190 Pragma Type_Invariant_Class . . . 97
2.191 Pragma Unchecked_Union . . . 98
2.192 Pragma Unevaluated_Use_Of_Old . . . 98
2.193 Pragma User_Aspect_Definition . . . 98
2.194 Pragma Unimplemented_Unit . . . 99
2.195 Pragma Universal_Aliasing . . . 99
2.196 Pragma Unmodified . . . 99
2.197 Pragma Unreferenced . . . 100
2.198 Pragma Unreferenced_Objects . . . 101
2.199 Pragma Unreserve_All_Interrupts . . . 101
2.200 Pragma Unsuppress . . . 101
2.201 Pragma Unused . . . 102
2.202 Pragma Use_VADS_Size . . . 102
2.203 Pragma Validity_Checks . . . 102
2.204 Pragma Volatile . . . 103
2.205 Pragma Volatile_Full_Access . . . 103
2.206 Pragma Volatile_Function . . . 104
2.207 Pragma Warning_As_Error . . . 104
2.208 Pragma Warnings . . . 105
2.209 Pragma Weak_External . . . 107
2.210 Pragma Wide_Character_Encoding . . . 108

3 Implementation Defined Aspects . . . 109

3.1 Aspect Abstract_State . . . 109
3.2 Aspect Always_Terminates . . . 109
3.3 Aspect Annotate . . . 109
3.4 Aspect Async_Readers . . . 110
3.5 Aspect Async_Writers . . . 110
3.6 Aspect Constant_After_Elaboration . . . 110
3.7 Aspect Contract_Cases . . . 110
3.8 Aspect Depends . . . 110
3.9 Aspect Default_Initial_Condition . . . 110
3.10 Aspect Dimension . . . 110
3.11 Aspect Dimension_System . . . 111
3.12 Aspect Disable_Controlled . . . 112

3.13 Aspect Effective_Reads ... 112
3.14 Aspect Effective_Writes ... 112
3.15 Aspect Exceptional_Cases ... 112
3.16 Aspect Exit_Cases ... 112
3.17 Aspect Extended_Access ... 112
3.18 Aspect Extensions_Visible ... 113
3.19 Aspect Favor_Top_Level ... 114
3.20 Aspect Ghost ... 114
3.21 Aspect Ghost_Predicate ... 114
3.22 Aspect Global ... 114
3.23 Aspect Initial_Condition ... 114
3.24 Aspect Initializes ... 114
3.25 Aspect Inline_Always ... 114
3.26 Aspect Invariant ... 114
3.27 Aspect Invariant'Class ... 114
3.28 Aspect Iterable ... 114
3.29 Aspect Linker_Section ... 115
3.30 Aspect Local_Restrictions ... 115
3.31 Aspect Lock_Free ... 116
3.32 Aspect Max_Queue_Length ... 116
3.33 Aspect No_Caching ... 116
3.34 Aspect No_Elaboration_Code_All ... 116
3.35 Aspect No_Inline ... 116
3.36 Aspect No_Raise ... 116
3.37 Aspect No_Tagged_Streams ... 117
3.38 Aspect No_Task_Parts ... 117
3.39 Aspect Object_Size ... 117
3.40 Aspect Obsolescent ... 117
3.41 Aspect Part_Of ... 117
3.42 Aspect Persistent_BSS ... 117
3.43 Aspect Potentially_Invalid ... 117
3.44 Aspect Predicate ... 117
3.45 Aspect Program_Exit ... 117
3.46 Aspect Pure_Function ... 117
3.47 Aspect Refined_Depends ... 118
3.48 Aspect Refined_Global ... 118
3.49 Aspect Refined_Post ... 118
3.50 Aspect Refined_State ... 118
3.51 Aspect Relaxed_Initialization ... 118
3.52 Aspect Remote_Access_Type ... 118
3.53 Aspect Scalar_Storage_Order ... 118
3.54 Aspect Secondary_Stack_Size ... 118
3.55 Aspect Shared ... 118
3.56 Aspect Side_Effects ... 118
3.57 Aspect Simple_Storage_Pool ... 118
3.58 Aspect Simple_Storage_Pool_Type ... 118
3.59 Aspect SPARK_Mode ... 119

3.60 Aspect Subprogram_Variant ... 119
3.61 Aspect Suppress_Debug_Info ... 119
3.62 Aspect Suppress_Initialization ... 119
3.63 Aspect Test_Case ... 119
3.64 Aspect Thread_Local_Storage ... 119
3.65 Aspect Universal_Aliasing ... 119
3.66 Aspect Unmodified ... 119
3.67 Aspect Unreferenced ... 119
3.68 Aspect Unreferenced_Objects ... 119
3.69 Aspect User_Aspect ... 119
3.70 Aspect Value_Size ... 120
3.71 Aspect Volatile_Full_Access ... 120
3.72 Aspect Volatile_Function ... 120
3.73 Aspect Warnings ... 120

4 Implementation Defined Attributes ... 121

4.1 Attribute Abort_Signal ... 121
4.2 Attribute Address_Size ... 121
4.3 Attribute Asm_Input ... 121
4.4 Attribute Asm_Output ... 121
4.5 Attribute Atomic_Always_Lock_Free ... 122
4.6 Attribute Bit ... 122
4.7 Attribute Bit_Position ... 122
4.8 Attribute Code_Address ... 122
4.9 Attribute Compiler_Version ... 123
4.10 Attribute Constrained ... 123
4.11 Attribute Default_Bit_Order ... 123
4.12 Attribute Default_Scalar_Storage_Order ... 123
4.13 Attribute Deref ... 123
4.14 Attribute Descriptor_Size ... 123
4.15 Attribute Elaborated ... 124
4.16 Attribute Elab_Body ... 124
4.17 Attribute Elab_Spec ... 124
4.18 Attribute Elab_Subp_Body ... 124
4.19 Attribute Emax ... 124
4.20 Attribute Enabled ... 125
4.21 Attribute Enum_Rep ... 125
4.22 Attribute Enum_Val ... 125
4.23 Attribute Epsilon ... 126
4.24 Attribute Fast_Math ... 126
4.25 Attribute Finalization_Size ... 126
4.26 Attribute Fixed_Value ... 126
4.27 Attribute From_Address ... 126
4.28 Attribute From_Any ... 127
4.29 Attribute Has_Access_Values ... 127
4.30 Attribute Has_Discriminants ... 127

4.31 Attribute Has_Tagged_Values ... 127
4.32 Attribute Img ... 127
4.33 Attribute Initialized ... 128
4.34 Attribute Integer_Value ... 128
4.35 Attribute Invalid_Value ... 128
4.36 Attribute Large ... 128
4.37 Attribute Library_Level ... 128
4.38 Attribute Loop_Entry ... 129
4.39 Attribute Machine_Size ... 129
4.40 Attribute Mantissa ... 129
4.41 Attribute Maximum_Alignment ... 129
4.42 Attribute Max_Integer_Size ... 129
4.43 Attribute Mechanism_Code ... 130
4.44 Attribute Null_Parameter ... 130
4.45 Attribute Object_Size ... 130
4.46 Attribute Old ... 131
4.47 Attribute Passed_By_Reference ... 131
4.48 Attribute Pool_Address ... 131
4.49 Attribute Range_Length ... 132
4.50 Attribute Restriction_Set ... 132
4.51 Attribute Result ... 133
4.52 Attribute Round ... 133
4.53 Attribute Safe_Emax ... 133
4.54 Attribute Safe_Large ... 133
4.55 Attribute Safe_Small ... 133
4.56 Attribute Scalar_Storage_Order ... 133
4.57 Attribute Simple_Storage_Pool ... 136
4.58 Attribute Small ... 136
4.59 Attribute Small_Denominator ... 137
4.60 Attribute Small_Numerator ... 137
4.61 Attribute Storage_Unit ... 137
4.62 Attribute Stub_Type ... 137
4.63 Attribute System_Allocator_Alignment ... 137
4.64 Attribute Target_Name ... 137
4.65 Attribute To_Address ... 138
4.66 Attribute To_Any ... 138
4.67 Attribute Type_Class ... 138
4.68 Attribute Type_Key ... 138
4.69 Attribute TypeCode ... 138
4.70 Attribute Unconstrained_Array ... 139
4.71 Attribute Universal_Literal_String ... 139
4.72 Attribute Unrestricted_Access ... 139
4.73 Attribute Update ... 142
4.74 Attribute Valid_Value ... 143
4.75 Attribute Valid_Scalars ... 143
4.76 Attribute VADS_Size ... 144
4.77 Attribute Value_Size ... 144

4.78 Attribute Wchar_T_Size 144
4.79 Attribute Word_Size 144

5 Standard and Implementation Defined Restrictions 145

5.1 Partition-Wide Restrictions 145
5.1.1 Immediate_Reclamation 145
5.1.2 Max_Asynchronous_Select_Nesting 145
5.1.3 Max_Entry_Queue_Length 145
5.1.4 Max_Protected_Entries 145
5.1.5 Max_Select_Alternatives 145
5.1.6 Max_Storage_At_Blocking 146
5.1.7 Max_Task_Entries 146
5.1.8 Max_Tasks 146
5.1.9 No_Abort_Statements 146
5.1.10 No_Access_Parameter_Allocators 146
5.1.11 No_Access_Subprograms 146
5.1.12 No_Allocators 146
5.1.13 No_Anonymous_Allocators 146
5.1.14 No_Asynchronous_Control 146
5.1.15 No_Calendar 146
5.1.16 No_Coextensions 146
5.1.17 No_Default_Initialization 147
5.1.18 No_Delay 147
5.1.19 No_Dependence 147
5.1.20 No_Direct_Boolean_Operators 147
5.1.21 No_Dispatch 147
5.1.22 No_Dispatching_Calls 147
5.1.23 No_Dynamic_Attachment 149
5.1.24 No_Dynamic_Priorities 149
5.1.25 No_Entry_Calls_In_Elaboration_Code 149
5.1.26 No_Enumeration_Maps 149
5.1.27 No_Exception_Handlers 149
5.1.28 No_Exception_Propagation 149
5.1.29 No_Exception_Registration 150
5.1.30 No_Exceptions 150
5.1.31 No_Finalization 150
5.1.32 No_Fixed_Point 150
5.1.33 No_Floating_Point 150
5.1.34 No_Implicit_Conditionals 150
5.1.35 No_Implicit_Dynamic_Code 151
5.1.36 No_Implicit_Heap_Allocations 151
5.1.37 No_Implicit_Protected_Object_Allocations 151
5.1.38 No_Implicit_Task_Allocations 151
5.1.39 No_Initialize_Scalars 151
5.1.40 No_IO 151

5.1.41 No_Local_Allocators 151
5.1.42 No_Local_Protected_Objects 151
5.1.43 No_Local_Tagged_Types 151
5.1.44 No_Local_Timing_Events 152
5.1.45 No_Long_Long_Integers 152
5.1.46 No_Multiple_Elaboration 152
5.1.47 No_Nested_Finalization 152
5.1.48 No_Protected_Type_Allocators 152
5.1.49 No_Protected_Types 152
5.1.50 No_Recursion 152
5.1.51 No_Reentrancy 152
5.1.52 No_Relative_Delay 152
5.1.53 No_Requeue_Statements 152
5.1.54 No_Secondary_Stack 153
5.1.55 No_Select_Statements 153
5.1.56 No_Specific_Termination_Handlers 153
5.1.57 No_Specification_of_Aspect 153
5.1.58 No_Standard_Allocators_After_Elaboration 153
5.1.59 No_Standard_Storage_Pools 153
5.1.60 No_Stream_Optimizations 153
5.1.61 No_Streams 153
5.1.62 No_Tagged_Type_Registration 154
5.1.63 No_Task_Allocators 154
5.1.64 No_Task_At_Interrupt_Priority 154
5.1.65 No_Task_Attributes_Package 154
5.1.66 No_Task_Hierarchy 154
5.1.67 No_Task_Termination 154
5.1.68 No_Tasking 154
5.1.69 No_Terminate_Alternatives 154
5.1.70 No_Unchecked_Access 155
5.1.71 No_Unchecked_Conversion 155
5.1.72 No_Unchecked_Deallocation 155
5.1.73 No_Use_Of_Attribute 155
5.1.74 No_Use_Of_Entity 155
5.1.75 No_Use_Of_Pragma 155
5.1.76 Pure_Barriers 155
5.1.77 Simple_Barriers 156
5.1.78 Static_Priorities 156
5.1.79 Static_Storage_Size 156
5.2 Program Unit Level Restrictions 156
5.2.1 No_Elaboration_Code 156
5.2.2 No_Dynamic_Accessibility_Checks 157
5.2.3 No_Dynamic_Sized_Objects 157
5.2.4 No_Entry_Queue 158
5.2.5 No_Implementation_Aspect_Specifications 158
5.2.6 No_Implementation_Attributes 158
5.2.7 No_Implementation_Identifiers 158

5.2.8 No_Implementation_Pragmas ... 158
5.2.9 No_Implementation_Restrictions ... 158
5.2.10 No_Implementation_Units ... 158
5.2.11 No_Implicit_Aliasing ... 158
5.2.12 No_Implicit_Loops ... 159
5.2.13 No_Obsolescent_Features ... 159
5.2.14 No_Wide_Characters ... 159
5.2.15 Static_Dispatch_Tables ... 159
5.2.16 SPARK_05 ... 159

6 Implementation Advice ... 160

6.1 RM 1.1.3(20): Error Detection ... 160
6.2 RM 1.1.3(31): Child Units ... 160
6.3 RM 1.1.5(12): Bounded Errors ... 160
6.4 RM 2.8(16): Pragmas ... 160
6.5 RM 2.8(17-19): Pragmas ... 161
6.6 RM 3.5.2(5): Alternative Character Sets ... 161
6.7 RM 3.5.4(28): Integer Types ... 162
6.8 RM 3.5.4(29): Integer Types ... 162
6.9 RM 3.5.5(8): Enumeration Values ... 162
6.10 RM 3.5.7(17): Float Types ... 162
6.11 RM 3.6.2(11): Multidimensional Arrays ... 163
6.12 RM 9.6(30-31): Duration'Small ... 163
6.13 RM 10.2.1(12): Consistent Representation ... 163
6.14 RM 11.4.1(19): Exception Information ... 163
6.15 RM 11.5(28): Suppression of Checks ... 164
6.16 RM 13.1 (21-24): Representation Clauses ... 164
6.17 RM 13.2(6-8): Packed Types ... 164
6.18 RM 13.3(14-19): Address Clauses ... 165
6.19 RM 13.3(29-35): Alignment Clauses ... 165
6.20 RM 13.3(42-43): Size Clauses ... 166
6.21 RM 13.3(50-56): Size Clauses ... 166
6.22 RM 13.3(71-73): Component Size Clauses ... 167
6.23 RM 13.4(9-10): Enumeration Representation Clauses ... 167
6.24 RM 13.5.1(17-22): Record Representation Clauses ... 167
6.25 RM 13.5.2(5): Storage Place Attributes ... 168
6.26 RM 13.5.3(7-8): Bit Ordering ... 168
6.27 RM 13.7(37): Address as Private ... 168
6.28 RM 13.7.1(16): Address Operations ... 168
6.29 RM 13.9(14-17): Unchecked Conversion ... 168
6.30 RM 13.11(23-25): Implicit Heap Usage ... 169
6.31 RM 13.11.2(17): Unchecked Deallocation ... 169
6.32 RM 13.13.2(1.6): Stream Oriented Attributes ... 169
6.33 RM A.1(52): Names of Predefined Numeric Types ... 170
6.34 RM A.3.2(49): `Ada.Characters.Handling` ... 170
6.35 RM A.4.4(106): Bounded-Length String Handling ... 170

6.36 RM A.5.2(46-47): Random Number Generation 170
6.37 RM A.10.7(23): `Get_Immediate` 171
6.38 RM A.18: `Containers` 171
6.39 RM B.1(39-41): Pragma `Export` 171
6.40 RM B.2(12-13): Package `Interfaces` 172
6.41 RM B.3(63-71): Interfacing with C 172
6.42 RM B.4(95-98): Interfacing with COBOL 173
6.43 RM B.5(22-26): Interfacing with Fortran 173
6.44 RM C.1(3-5): Access to Machine Operations 174
6.45 RM C.1(10-16): Access to Machine Operations 174
6.46 RM C.3(28): Interrupt Support 175
6.47 RM C.3.1(20-21): Protected Procedure Handlers 175
6.48 RM C.3.2(25): Package `Interrupts` 175
6.49 RM C.4(14): Pre-elaboration Requirements 175
6.50 RM C.5(8): Pragma `Discard_Names` 175
6.51 RM C.7.2(30): The Package Task_Attributes 175
6.52 RM D.3(17): Locking Policies 176
6.53 RM D.4(16): Entry Queuing Policies 176
6.54 RM D.6(9-10): Preemptive Abort 176
6.55 RM D.7(21): Tasking Restrictions 176
6.56 RM D.8(47-49): Monotonic Time 176
6.57 RM E.5(28-29): Partition Communication Subsystem 177
6.58 RM F(7): COBOL Support 177
6.59 RM F.1(2): Decimal Radix Support 177
6.60 RM G: Numerics 177
6.61 RM G.1.1(56-58): Complex Types 178
6.62 RM G.1.2(49): Complex Elementary Functions 178
6.63 RM G.2.4(19): Accuracy Requirements 179
6.64 RM G.2.6(15): Complex Arithmetic Accuracy 179
6.65 RM H.6(15/2): Pragma Partition_Elaboration_Policy 179

7 Implementation Defined Characteristics 180

8 Intrinsic Subprograms 199

8.1 Intrinsic Operators 199
8.2 Compilation_ISO_Date 199
8.3 Compilation_Date 199
8.4 Compilation_Time 199
8.5 Enclosing_Entity 200
8.6 Exception_Information 200
8.7 Exception_Message 200
8.8 Exception_Name 200
8.9 File 200
8.10 Line 200
8.11 Shifts and Rotates 201
8.12 Source_Location 201

9 Representation Clauses and Pragmas 202
9.1 Alignment Clauses 202
9.2 Size Clauses 203
9.3 Storage_Size Clauses 204
9.4 Size of Variant Record Objects 205
9.5 Biased Representation 207
9.6 Value_Size and Object_Size Clauses 207
9.7 Component_Size Clauses 210
9.8 Bit_Order Clauses 211
9.9 Effect of Bit_Order on Byte Ordering 212
9.10 Pragma Pack for Arrays 216
9.11 Pragma Pack for Records 218
9.12 Record Representation Clauses 219
9.13 Handling of Records with Holes 220
9.14 Enumeration Clauses 221
9.15 Address Clauses 222
9.16 Use of Address Clauses for Memory-Mapped I/O 227
9.17 Effect of Convention on Representation 227
9.18 Conventions and Anonymous Access Types 228
9.19 Determining the Representations chosen by GNAT 230

10 Standard Library Routines 233

11 The Implementation of Standard I/O 244
11.1 Standard I/O Packages 244
11.2 FORM Strings 245
11.3 Direct_IO 245
11.4 Sequential_IO 245
11.5 Text_IO 246
11.5.1 Stream Pointer Positioning 247
11.5.2 Reading and Writing Non-Regular Files 247
11.5.3 Get_Immediate 248
11.5.4 Treating Text_IO Files as Streams 248
11.5.5 Text_IO Extensions 248
11.5.6 Text_IO Facilities for Unbounded Strings 248
11.6 Wide_Text_IO 249
11.6.1 Stream Pointer Positioning 251
11.6.2 Reading and Writing Non-Regular Files 252
11.7 Wide_Wide_Text_IO 252
11.7.1 Stream Pointer Positioning 253
11.7.2 Reading and Writing Non-Regular Files 253
11.8 Stream_IO 254
11.9 Text Translation 254
11.10 Shared Files 254
11.11 Filenames encoding 255

11.12 File content encoding 255
11.13 Open Modes 256
11.14 Operations on C Streams 256
11.15 Interfacing to C Streams 259

12 The GNAT Library 262

12.1 `Ada.Characters.Latin_9` (`a-chlat9.ads`) 262
12.2 `Ada.Characters.Wide_Latin_1` (`a-cwila1.ads`) 262
12.3 `Ada.Characters.Wide_Latin_9` (`a-cwila9.ads`) 262
12.4 `Ada.Characters.Wide_Wide_Latin_1` (`a-chzla1.ads`) 262
12.5 `Ada.Characters.Wide_Wide_Latin_9` (`a-chzla9.ads`) 263
12.6 `Ada.Containers.Bounded_Holders` (`a-coboho.ads`) 263
12.7 `Ada.Command_Line.Environment` (`a-colien.ads`) 263
12.8 `Ada.Command_Line.Remove` (`a-colire.ads`) 263
12.9 `Ada.Command_Line.Response_File` (`a-clrefi.ads`) 263
12.10 `Ada.Direct_IO.C_Streams` (`a-diocst.ads`) 263
12.11 `Ada.Exceptions.Is_Null_Occurrence` (`a-einuoc.ads`) 263
12.12 `Ada.Exceptions.Last_Chance_Handler` (`a-elchha.ads`) 263
12.13 `Ada.Exceptions.Traceback` (`a-exctra.ads`) 264
12.14 `Ada.Sequential_IO.C_Streams` (`a-siocst.ads`) 264
12.15 `Ada.Streams.Stream_IO.C_Streams` (`a-ssicst.ads`) 264
12.16 `Ada.Strings.Unbounded.Text_IO` (`a-suteio.ads`) 264
12.17 `Ada.Strings.Wide_Unbounded.Wide_Text_IO` (`a-swuwti.ads`) 264
12.18 `Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Text_IO` (`a-szuzti.ads`) 264
12.19 `Ada.Task_Initialization` (`a-tasini.ads`) 264
12.20 `Ada.Text_IO.C_Streams` (`a-tiocst.ads`) 264
12.21 `Ada.Text_IO.Reset_Standard_Files` (`a-tirsfi.ads`) 264
12.22 `Ada.Wide_Characters.Unicode` (`a-wichun.ads`) 265
12.23 `Ada.Wide_Text_IO.C_Streams` (`a-wtcstr.ads`) 265
12.24 `Ada.Wide_Text_IO.Reset_Standard_Files` (`a-wrstfi.ads`) 265
12.25 `Ada.Wide_Wide_Characters.Unicode` (`a-zchuni.ads`) 265
12.26 `Ada.Wide_Wide_Text_IO.C_Streams` (`a-ztcstr.ads`) 265
12.27 `Ada.Wide_Wide_Text_IO.Reset_Standard_Files` (`a-zrstfi.ads`) 265
12.28 `GNAT.Altivec` (`g-altive.ads`) 265
12.29 `GNAT.Altivec.Conversions` (`g-altcon.ads`) 265
12.30 `GNAT.Altivec.Vector_Operations` (`g-alveop.ads`) 266
12.31 `GNAT.Altivec.Vector_Types` (`g-alvety.ads`) 266
12.32 `GNAT.Altivec.Vector_Views` (`g-alvevi.ads`) 266
12.33 `GNAT.Array_Split` (`g-arrspl.ads`) 266
12.34 `GNAT.AWK` (`g-awk.ads`) 266
12.35 `GNAT.Binary_Search` (`g-binsea.ads`) 266
12.36 `GNAT.Bind_Environment` (`g-binenv.ads`) 266
12.37 `GNAT.Branch_Prediction` (`g-brapre.ads`) 266

12.38 GNAT.Bounded_Buffers (g-boubuf.ads) 266
12.39 GNAT.Bounded_Mailboxes (g-boumai.ads)................ 266
12.40 GNAT.Bubble_Sort (g-bubsor.ads) 267
12.41 GNAT.Bubble_Sort_A (g-busora.ads)..................... 267
12.42 GNAT.Bubble_Sort_G (g-busorg.ads)..................... 267
12.43 GNAT.Byte_Order_Mark (g-byorma.ads) 267
12.44 GNAT.Byte_Swapping (g-bytswa.ads)..................... 267
12.45 GNAT.C_Time (g-c_time.ads) 267
12.46 GNAT.Calendar (g-calend.ads)........................... 267
12.47 GNAT.Calendar.Time_IO (g-catiio.ads) 267
12.48 GNAT.CRC32 (g-crc32.ads) 267
12.49 GNAT.Case_Util (g-casuti.ads) 267
12.50 GNAT.CGI (g-cgi.ads) 268
12.51 GNAT.CGI.Cookie (g-cgicoo.ads) 268
12.52 GNAT.CGI.Debug (g-cgideb.ads) 268
12.53 GNAT.Command_Line (g-comlin.ads)...................... 268
12.54 GNAT.Compiler_Version (g-comver.ads) 268
12.55 GNAT.Ctrl_C (g-ctrl_c.ads) 268
12.56 GNAT.Current_Exception (g-curexc.ads)................ 268
12.57 GNAT.Debug_Pools (g-debpoo.ads) 268
12.58 GNAT.Debug_Utilities (g-debuti.ads) 268
12.59 GNAT.Decode_String (g-decstr.ads)..................... 269
12.60 GNAT.Decode_UTF8_String (g-deutst.ads)............... 269
12.61 GNAT.Directory_Operations (g-dirope.ads) 269
12.62 GNAT.Directory_Operations.Iteration (g-diopit.ads).. 269
12.63 GNAT.Dynamic_HTables (g-dynhta.ads) 269
12.64 GNAT.Dynamic_Tables (g-dyntab.ads) 269
12.65 GNAT.Encode_String (g-encstr.ads)..................... 269
12.66 GNAT.Encode_UTF8_String (g-enutst.ads)............... 269
12.67 GNAT.Exception_Actions (g-excact.ads)................ 270
12.68 GNAT.Exception_Traces (g-exctra.ads) 270
12.69 GNAT.Exceptions (g-except.ads) 270
12.70 GNAT.Expect (g-expect.ads) 270
12.71 GNAT.Expect.TTY (g-exptty.ads) 270
12.72 GNAT.Float_Control (g-flocon.ads)..................... 270
12.73 GNAT.Formatted_String (g-forstr.ads) 270
12.74 GNAT.Generic_Fast_Math_Functions (g-gfmafu.ads) 270
12.75 GNAT.Heap_Sort (g-heasor.ads) 271
12.76 GNAT.Heap_Sort_A (g-hesora.ads) 271
12.77 GNAT.Heap_Sort_G (g-hesorg.ads) 271
12.78 GNAT.HTable (g-htable.ads)............................. 271
12.79 GNAT.IO (g-io.ads) 271
12.80 GNAT.IO_Aux (g-io_aux.ads)............................. 271
12.81 GNAT.Lock_Files (g-locfil.ads) 271
12.82 GNAT.MBBS_Discrete_Random (g-mbdira.ads) 272
12.83 GNAT.MBBS_Float_Random (g-mbflra.ads)................ 272
12.84 GNAT.MD5 (g-md5.ads)................................... 272

12.85 GNAT.Memory_Dump (g-memdum.ads) 272
12.86 GNAT.Most_Recent_Exception (g-moreex.ads) 272
12.87 GNAT.OS_Lib (g-os_lib.ads) 272
12.88 GNAT.Perfect_Hash_Generators (g-pehage.ads) 272
12.89 GNAT.Random_Numbers (g-rannum.ads) 272
12.90 GNAT.Regexp (g-regexp.ads) 273
12.91 GNAT.Registry (g-regist.ads) 273
12.92 GNAT.Regpat (g-regpat.ads) 273
12.93 GNAT.Rewrite_Data (g-rewdat.ads) 273
12.94 GNAT.Secondary_Stack_Info (g-sestin.ads) 273
12.95 GNAT.Semaphores (g-semaph.ads) 273
12.96 GNAT.Serial_Communications (g-sercom.ads) 273
12.97 GNAT.SHA1 (g-sha1.ads) 273
12.98 GNAT.SHA224 (g-sha224.ads) 273
12.99 GNAT.SHA256 (g-sha256.ads) 274
12.100 GNAT.SHA384 (g-sha384.ads) 274
12.101 GNAT.SHA512 (g-sha512.ads) 274
12.102 GNAT.Signals (g-signal.ads) 274
12.103 GNAT.Sockets (g-socket.ads) 274
12.104 GNAT.Source_Info (g-souinf.ads) 274
12.105 GNAT.Spelling_Checker (g-speche.ads) 274
12.106 GNAT.Spelling_Checker_Generic (g-spchge.ads) 274
12.107 GNAT.Spitbol.Patterns (g-spipat.ads) 274
12.108 GNAT.Spitbol (g-spitbo.ads) 275
12.109 GNAT.Spitbol.Table_Boolean (g-sptabo.ads) 275
12.110 GNAT.Spitbol.Table_Integer (g-sptain.ads) 275
12.111 GNAT.Spitbol.Table_VString (g-sptavs.ads) 275
12.112 GNAT.SSE (g-sse.ads) 275
12.113 GNAT.SSE.Vector_Types (g-ssvety.ads) 275
12.114 GNAT.String_Hash (g-strhas.ads) 275
12.115 GNAT.Strings (g-string.ads) 275
12.116 GNAT.String_Split (g-strspl.ads) 275
12.117 GNAT.Table (g-table.ads) 276
12.118 GNAT.Task_Lock (g-tasloc.ads) 276
12.119 GNAT.Time_Stamp (g-timsta.ads) 276
12.120 GNAT.Threads (g-thread.ads) 276
12.121 GNAT.Traceback (g-traceb.ads) 276
12.122 GNAT.Traceback.Symbolic (g-trasym.ads) 276
12.123 GNAT.UTF_32 (g-utf_32.ads) 276
12.124 GNAT.UTF_32_Spelling_Checker (g-u3spch.ads) 276
12.125 GNAT.Wide_Spelling_Checker (g-wispch.ads) 277
12.126 GNAT.Wide_String_Split (g-wistsp.ads) 277
12.127 GNAT.Wide_Wide_Spelling_Checker (g-zspche.ads) 277
12.128 GNAT.Wide_Wide_String_Split (g-zistsp.ads) 277
12.129 Interfaces.C.Extensions (i-cexten.ads) 277
12.130 Interfaces.C.Streams (i-cstrea.ads) 277
12.131 Interfaces.Packed_Decimal (i-pacdec.ads) 277

12.132 Interfaces.VxWorks (i-vxwork.ads) 277
12.133 Interfaces.VxWorks.IO (i-vxwoio.ads) 277
12.134 System.Address_Image (s-addima.ads) 277
12.135 System.Assertions (s-assert.ads) 278
12.136 System.Atomic_Counters (s-atocou.ads) 278
12.137 System.Memory (s-memory.ads) 278
12.138 System.Multiprocessors (s-multip.ads) 278
12.139 System.Multiprocessors.Dispatching_Domains (s-mudido.ads) .. 278
12.140 System.Partition_Interface (s-parint.ads) 278
12.141 System.Pool_Global (s-pooglo.ads) 278
12.142 System.Pool_Local (s-pooloc.ads) 279
12.143 System.Restrictions (s-restri.ads) 279
12.144 System.Rident (s-rident.ads) 279
12.145 System.Strings.Stream_Ops (s-ststop.ads) 279
12.146 System.Unsigned_Types (s-unstyp.ads) 279
12.147 System.Wch_Cnv (s-wchcnv.ads) 279
12.148 System.Wch_Con (s-wchcon.ads) 279

13 Interfacing to Other Languages 280

13.1 Interfacing to C 280
13.2 Interfacing to C++ 281
13.3 Interfacing to COBOL 284
13.4 Interfacing to Fortran 284
13.5 Interfacing to non-GNAT Ada code 284

14 Specialized Needs Annexes 286

15 Implementation of Specific Ada Features .. 287

15.1 Machine Code Insertions 287
15.2 GNAT Implementation of Tasking 289
15.2.1 Mapping Ada Tasks onto the Underlying Kernel Threads .. 289
15.2.2 Ensuring Compliance with the Real-Time Annex 290
15.2.3 Support for Locking Policies 290
15.3 GNAT Implementation of Shared Passive Packages 291
15.4 Code Generation for Array Aggregates 292
15.4.1 Static constant aggregates with static bounds 292
15.4.2 Constant aggregates with unconstrained nominal types .. 293
15.4.3 Aggregates with static bounds 293
15.4.4 Aggregates with nonstatic bounds 293
15.4.5 Aggregates in assignment statements 293
15.5 The Size of Discriminated Records with Default Discriminants .. 294
15.6 Image Values For Nonscalar Types 295
15.7 Strict Conformance to the Ada Reference Manual 295

16 Implementation of Ada 2022 Features 297

17 GNAT language extensions 330

17.1 How to activate the extended GNAT Ada superset 330
17.2 Curated Extensions 330
17.2.1 Local Declarations Without Block 330
17.2.2 Deep delta Aggregates 332
17.2.2.1 Syntax 332
17.2.2.2 Legality Rules 333
17.2.2.3 Dynamic Semantics 333
17.2.2.4 Examples 334
17.2.3 Fixed lower bounds for array types and subtypes 334
17.2.4 Prefixed-view notation for calls to primitive subprograms of untagged types 335
17.2.5 Expression defaults for generic formal functions 336
17.2.6 String interpolation 336
17.2.7 Constrained attribute for generic objects 337
17.2.8 `Static` aspect on intrinsic functions 337
17.2.9 First Controlling Parameter 338
17.2.10 Generalized Finalization 339
17.2.10.1 Finalizable tagged types 341
17.2.10.2 Composite types 341
17.2.10.3 Interoperability with controlled types 341
17.3 Experimental Language Extensions 341
17.3.1 Conditional when constructs 341
17.3.2 Implicit With 342
17.3.3 Storage Model 342
17.3.3.1 Aspect Storage_Model_Type 343
17.3.3.2 Aspect Designated_Storage_Model 345
17.3.3.3 Legacy Storage Pools 346
17.3.4 Attribute Super 347
17.3.5 Simpler Accessibility Model 347
17.3.5.1 Stand-alone objects 348
17.3.5.2 Subprogram parameters 348
17.3.5.3 Function results 350
17.3.6 Case pattern matching 352
17.3.7 Mutably Tagged Types with Size'Class Aspect 354
17.3.8 No_Raise aspect 356
17.3.9 Inference of Dependent Types in Generic Instantiations .. 356
17.3.10 External_Initialization Aspect 357
17.3.11 Finally construct 358
17.3.11.1 Syntax 358
17.3.11.2 Legality Rules 358
17.3.11.3 Dynamic Semantics 358
17.3.12 Continue statement 359
17.3.13 Destructors 359

17.3.14 Structural Generic Instantiation 360
17.3.14.1 Syntax 360
17.3.14.2 Legality Rules 360
17.3.14.3 Static Semantics 360

18 Security Hardening Features 364

18.1 Register Scrubbing 364
18.2 Stack Scrubbing 364
18.3 Hardened Conditionals 366
18.4 Hardened Booleans 368
18.5 Control Flow Redundancy 369

19 Obsolescent Features 372

19.1 PolyORB 372
19.2 pragma No_Run_Time 372
19.3 pragma Ravenscar 372
19.4 pragma Restricted_Run_Time 372
19.5 pragma Task_Info 372
19.6 package System.Task_Info (`s-tasinf.ads`) 373

20 Compatibility and Porting Guide 374

20.1 Writing Portable Fixed-Point Declarations 374
20.2 Compatibility with Ada 83 375
20.2.1 Legal Ada 83 programs that are illegal in Ada 95 375
20.2.2 More deterministic semantics 377
20.2.3 Changed semantics 377
20.2.4 Other language compatibility issues 377
20.3 Compatibility between Ada 95 and Ada 2005 378
20.4 Implementation-dependent characteristics 379
20.4.1 Implementation-defined pragmas 379
20.4.2 Implementation-defined attributes 379
20.4.3 Libraries 379
20.4.4 Elaboration order 379
20.4.5 Target-specific aspects 380
20.5 Compatibility with Other Ada Systems 380
20.6 Representation Clauses 381
20.7 Compatibility with HP Ada 83 382

21 GNU Free Documentation License 383

Index 390

‘GNAT, The GNU Ada Development Environment’

GCC version 16.0.1
AdaCore

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 363, documents GNAT extensions aimed at security hardening.
* [Obsolescent Features], page 371, 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 373, 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 382, 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.

Note that, as with the `if` statement to which it is equivalent, the type of the expression is either `Standard.Boolean`, or any type derived from this standard type.

Assert checks can be either checked or ignored. By default they are ignored. They will be checked if either the command line switch '-gnata' is used, or if an `Assertion_Policy` or `Check_Policy` pragma is used to enable `Assert_Checks`.

If assertions are ignored, then there is no run-time effect (and in particular, any side effects from the expression will not occur at run time). (The expression is still analyzed at compile time, and may cause types to be frozen if they are mentioned here for the first time).

If assertions are checked, then the given expression is tested, and if it is `False` then `System.Assertions.Raise_Assert_Failure` is called which results in the raising of `Assert_Failure` with the given message.

You should generally avoid side effects in the expression arguments of this pragma, because these side effects will turn on and off with the setting of the assertions mode, resulting in assertions that have an effect on the program. However, the expressions are analyzed for semantic correctness whether or not assertions are enabled, so turning assertions on and off cannot affect the legality of a program.

Note that the implementation defined policy `DISABLE`, given in a pragma `Assertion_Policy`, can be used to suppress this semantic analysis.

Note: this is a standard language-defined pragma in versions of Ada from 2005 on. In GNAT, it is implemented in all versions of Ada, and the DISABLE policy is an implementation-defined addition.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.15 Pragma Assert_And_Cut

Syntax:

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

The effect of this pragma is identical to that of pragma `Assert`, except that in an `Assertion_Policy` pragma, the identifier `Assert_And_Cut` is used to control whether it is ignored or checked (or disabled).

The intention is that this be used within a subprogram when the given test expresion sums up all the work done so far in the subprogram, so that the rest of the subprogram can be verified (informally or formally) using only the entry preconditions, and the expression in this pragma. This allows dividing up a subprogram into sections for the purposes of testing or formal verification. The pragma also serves as useful documentation.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.16 Pragma Assertion_Level

Syntax:

```
pragma Assertion_Level (LEVEL_IDENTIFIER
```

```
                          [, 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
```

```
POLICY_IDENTIFIER ::= Check | Disable | Ignore | Suppressible
```

This is a standard Ada 2012 pragma that is available as an implementation-defined pragma in earlier versions of Ada. The assertion kinds `RM_ASSERTION_KIND` are those defined in the Ada standard. The assertion kinds `ID_ASSERTION_KIND` are implementation defined additions recognized by the GNAT compiler.

Additionally the pragma can apply to an assertion level defined by the `Assertion_Level` pragma. For more details see the SPARK 2014 Reference Manual, section 11.4.2.

The pragma applies in both cases to pragmas and aspects with matching names, e.g. `Pre` applies to the Pre aspect, and `Precondition` applies to both the `Precondition` pragma and the aspect `Precondition`. Note that the identifiers for pragmas Pre_Class and Post_Class are Pre'Class and Post'Class (not Pre_Class and Post_Class), since these pragmas are intended to be identical to the corresponding aspects.

If the policy is `CHECK`, then assertions are enabled, i.e. the corresponding pragma or aspect is activated. If the policy is `IGNORE`, then assertions are ignored, i.e. the corresponding pragma or aspect is deactivated. This pragma overrides the effect of the '-gnata' switch on the command line. If the policy is `SUPPRESSIBLE`, then assertions are enabled by default, however, if the '-gnatp' switch is specified all assertions are ignored.

The implementation defined policy `DISABLE` is like `IGNORE` except that it completely disables semantic checking of the corresponding pragma or aspect. This is useful when the pragma or aspect argument references subprograms in a with'ed package which is replaced by a dummy package for the final build.

The implementation defined assertion kind `Assertions` applies to all assertion kinds. The form with no assertion kind given implies this choice, so it applies to all assertion kinds (RM defined, and implementation defined).

The implementation defined assertion kind `Statement_Assertions` applies to `Assert`, `Assert_And_Cut`, `Assume`, `Loop_Invariant`, and `Loop_Variant`.

2.18 Pragma Assume

Syntax:

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

The effect of this pragma is identical to that of pragma `Assert`, except that in an `Assertion_Policy` pragma, the identifier `Assume` is used to control whether it is ignored or checked (or disabled).

The intention is that this be used for assumptions about the external environment. So you cannot expect to verify formally or informally that the condition is met, this must be established by examining things outside the program itself. For example, we may have code that depends on the size of `Long_Long_Integer` being at least 64. So we could write:

```
pragma Assume (Long_Long_Integer'Size >= 64);
```

This assumption cannot be proved from the program itself, but it acts as a useful run-time check that the assumption is met, and documents the need to ensure that it is met by reference to information outside the program.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.19 Pragma Assume_No_Invalid_Values

Syntax:

```
pragma Assume_No_Invalid_Values (On | Off);
```

This is a configuration pragma that controls the assumptions made by the compiler about the occurrence of invalid representations (invalid values) in the code.

The default behavior (corresponding to an Off argument for this pragma), is to assume that values may in general be invalid unless the compiler can prove they are valid. Consider the following example:

```
V1 : Integer range 1 .. 10;
V2 : Integer range 11 .. 20;
...
for J in V2 .. V1 loop
   ...
end loop;
```

if V1 and V2 have valid values, then the loop is known at compile time not to execute since the lower bound must be greater than the upper bound. However in default mode, no such assumption is made, and the loop may execute. If `Assume_No_Invalid_Values (On)` is given, the compiler will assume that any occurrence of a variable other than in an explicit `'Valid` test always has a valid value, and the loop above will be optimized away.

The use of `Assume_No_Invalid_Values (On)` is appropriate if you know your code is free of uninitialized variables and other possible sources of invalid representations, and may result in more efficient code. A program that accesses an invalid representation with this pragma in effect is erroneous, so no guarantees can be made about its behavior.

It is peculiar though permissible to use this pragma in conjunction with validity checking (-gnatVa). In such cases, accessing invalid values will generally give an exception, though formally the program is erroneous so there are no guarantees that this will always be the case, and it is recommended that these two options not be used together.

2.20 Pragma Async_Readers

Syntax:

```
pragma Async_Readers [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Async_Readers` in the SPARK 2014 Reference Manual, section 7.1.2.

2.21 Pragma Async_Writers

Syntax:

```
pragma Async_Writers [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Async_Writers` in the SPARK 2014 Reference Manual, section 7.1.2.

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           |
```

```
Type_Invariant'Class |
Invariant'Class
```

This pragma is similar to the predefined pragma `Assert` except that an extra identifier argument is present. In conjunction with pragma `Check_Policy`, this can be used to define groups of assertions that can be independently controlled. The identifier `Assertion` is special, it refers to the normal set of pragma `Assert` statements.

Checks introduced by this pragma are normally deactivated by default. They can be activated either by the command line option '-gnata', which turns on all checks, or individually controlled using pragma `Check_Policy`.

The identifiers `Assertions` and `Statement_Assertions` are not permitted as check kinds, since this would cause confusion with the use of these identifiers in `Assertion_Policy` and `Check_Policy` pragmas, where they are used to refer to sets of assertions.

2.25 Pragma Check_Float_Overflow

Syntax:

```
pragma Check_Float_Overflow;
```

In Ada, the predefined floating-point types (`Short_Float`, `Float`, `Long_Float`, `Long_Long_Float`) are defined to be 'unconstrained'. This means that even though each has a well-defined base range, an operation that delivers a result outside this base range is not required to raise an exception. This implementation permission accommodates the notion of infinities in IEEE floating-point, and corresponds to the efficient execution mode on most machines. GNAT will not raise overflow exceptions on these machines; instead it will generate infinities and NaN's as defined in the IEEE standard.

Generating infinities, although efficient, is not always desirable. Often the preferable approach is to check for overflow, even at the (perhaps considerable) expense of run-time performance. This can be accomplished by defining your own constrained floating-point subtypes – i.e., by supplying explicit range constraints – and indeed such a subtype can have the same base range as its base type. For example:

```
subtype My_Float is Float range Float'Range;
```

Here `My_Float` has the same range as `Float` but is constrained, so operations on `My_Float` values will be checked for overflow against this range.

This style will achieve the desired goal, but it is often more convenient to be able to simply use the standard predefined floating-point types as long as overflow checking could be guaranteed. The `Check_Float_Overflow` configuration pragma achieves this effect. If a unit is compiled subject to this configuration pragma, then all operations on predefined floating-point types including operations on base types of these floating-point types will be treated as though those types were constrained, and overflow checks will be generated. The `Constraint_Error` exception is raised if the result is out of range.

This mode can also be set by use of the compiler switch '-gnateF'.

2.26 Pragma Check_Name

Syntax:

```
pragma Check_Name (check_name_IDENTIFIER);
```

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.

If the first argument is Debug, then the policy applies to Debug pragmas, disabling their effect if the policy is `OFF`, `DISABLE`, or `IGNORE`, and allowing them to execute with normal semantics if the policy is `ON` or `CHECK`. In addition if the policy is `DISABLE`, then the procedure call in `Debug` pragmas will be totally ignored and not analyzed semantically.

Finally the first argument may be some other identifier than the above possibilities, in which case it controls a set of named assertions that can be checked using pragma `Check`. For example, if the pragma:

```
pragma Check_Policy (Critical_Error, OFF);
```

is given, then subsequent `Check` pragmas whose first argument is also `Critical_Error` will be disabled.

The check policy is `OFF` to turn off corresponding checks, and `ON` to turn on corresponding checks. The default for a set of checks for which no `Check_Policy` is given is `OFF` unless the compiler switch '-gnata' is given, which turns on all checks by default.

The check policy settings `CHECK` and `IGNORE` are recognized as synonyms for `ON` and `OFF`. These synonyms are provided for compatibility with the standard `Assertion_Policy` pragma. The check policy setting `DISABLE` causes the second argument of a corresponding `Check` pragma to be completely ignored and not analyzed.

2.28 Pragma Comment

Syntax:

```
pragma Comment (static_string_EXPRESSION);
```

This is almost identical in effect to pragma `Ident`. It allows the placement of a comment into the object file and hence into the executable file if the operating system permits such usage. The difference is that `Comment`, unlike `Ident`, has no limitations on placement of the pragma (it can be placed anywhere in the main source unit), and if more than one pragma is used, all comments are retained.

2.29 Pragma Common_Object

Syntax:

```
pragma Common_Object (
     [Internal =>] LOCAL_NAME
  [, [External =>] EXTERNAL_SYMBOL]
  [, [Size     =>] EXTERNAL_SYMBOL] );

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma enables the shared use of variables stored in overlaid linker areas corresponding to the use of `COMMON` in Fortran. The single object `LOCAL_NAME` is assigned to the area designated by the `External` argument. You may define a record to correspond to a series of fields. The `Size` argument is syntax checked in GNAT, but otherwise ignored.

`Common_Object` is not supported on all platforms. If no support is available, then the code generator will issue a message indicating that the necessary attribute for implementation of this pragma is not available.

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).

If the Name parameter is present, type_LOCAL_NAME must refer to a local record or array type, and the specified alignment choice applies to the specified type. The use of Component_Alignment together with a pragma Pack causes the Component_Alignment pragma to be ignored. The use of Component_Alignment together with a record representation clause is only effective for fields not specified by the representation clause.

If the Name parameter is absent, the pragma can be used as either a configuration pragma, in which case it applies to one or more units in accordance with the normal rules for configuration pragmas, or it can be used within a declarative part, in which case it applies to types that are declared within this declarative part, or within any nested scope within this declarative part. In either case it specifies the alignment to be applied to any record or array type which has otherwise standard representation.

If the alignment for a record or array type is not specified (using pragma Pack, pragma Component_Alignment, or a record rep clause), the GNAT uses the default alignment as described previously.

2.35 Pragma Constant_After_Elaboration

Syntax:

```
pragma Constant_After_Elaboration [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect Constant_After_Elaboration in the SPARK 2014 Reference Manual, section 3.3.1.

2.36 Pragma Contract_Cases

Syntax:

```
pragma Contract_Cases (CONTRACT_CASE {, CONTRACT_CASE});

CONTRACT_CASE ::= CASE_GUARD => CONSEQUENCE

CASE_GUARD ::= boolean_EXPRESSION | others

CONSEQUENCE ::= boolean_EXPRESSION
```

The Contract_Cases pragma allows defining fine-grain specifications that can complement or replace the contract given by a precondition and a postcondition. Additionally, the Contract_Cases pragma can be used by testing and formal verification tools. The compiler checks its validity and, depending on the assertion policy at the point of declaration of the pragma, it may insert a check in the executable. For code generation, the contract cases

```
pragma Contract_Cases (
  Cond1 => Pred1,
  Cond2 => Pred2);
```

are equivalent to

```
C1 : constant Boolean := Cond1;  --  evaluated at subprogram entry
C2 : constant Boolean := Cond2;  --  evaluated at subprogram entry
pragma Precondition ((C1 and not C2) or (C2 and not C1));
pragma Postcondition (if C1 then Pred1);
pragma Postcondition (if C2 then Pred2);
```

The precondition ensures that one and only one of the case guards is satisfied on entry to the subprogram. The postcondition ensures that for the case guard that was True on entry, the corresponding consequence is True on exit. Other consequence expressions are not evaluated.

A precondition `P` and postcondition `Q` can also be expressed as contract cases:

```
pragma Contract_Cases (P => Q);
```

The placement and visibility rules for `Contract_Cases` pragmas are identical to those described for preconditions and postconditions.

The compiler checks that boolean expressions given in case guards and consequences are valid, where the rules for case guards are the same as the rule for an expression in `Precondition` and the rules for consequences are the same as the rule for an expression in `Postcondition`. In particular, attributes `'Old` and `'Result` can only be used within consequence expressions. The case guard for the last contract case may be `others`, to denote any case not captured by the previous cases. The following is an example of use within a package spec:

```
package Math_Functions is
   ...
   function Sqrt (Arg : Float) return Float;
   pragma Contract_Cases (((Arg in 0.0 .. 99.0) => Sqrt'Result < 10.0,
                           Arg >= 100.0         => Sqrt'Result >= 10.0,
                           others               => Sqrt'Result = 0.0));
   ...
end Math_Functions;
```

The meaning of contract cases is that only one case should apply at each call, as determined by the corresponding case guard evaluating to True, and that the consequence for this case should hold when the subprogram returns.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.37 Pragma Convention_Identifier

Syntax:

```
pragma Convention_Identifier (
         [Name =>]       IDENTIFIER,
         [Convention =>] convention_IDENTIFIER);
```

This pragma provides a mechanism for supplying synonyms for existing convention identifiers. The `Name` identifier can subsequently be used as a synonym for the given convention in other pragmas (including for example pragma `Import` or another `Convention_Identifier` pragma). As an example of the use of this, suppose you had legacy code which used Fortran77 as the identifier for Fortran. Then the pragma:

```
pragma Convention_Identifier (Fortran77, Fortran);
```

would allow the use of the convention identifier `Fortran77` in subsequent code, avoiding the need to modify the sources. As another example, you could use this to parameterize convention requirements according to systems. Suppose you needed to use `Stdcall` on windows systems, and `C` on some other system, then you could define a convention identifier

Library and use a single Convention_Identifier pragma to specify which convention would be used system-wide.

2.38 Pragma CPP_Class

Syntax:

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

The argument denotes an entity in the current declarative region that is declared as a record type. It indicates that the type corresponds to an externally declared C++ class type, and is to be laid out the same way that C++ would lay out the type. If the C++ class has virtual primitives then the record must be declared as a tagged record type.

Types for which CPP_Class is specified do not have assignment or equality operators defined (such operations can be imported or declared as subprograms as required). Initialization is allowed only by constructor functions (see pragma CPP_Constructor). Such types are implicitly limited if not explicitly declared as limited or derived from a limited type, and an error is issued in that case.

See [Interfacing to C++], page 281, for related information.

Note: Pragma CPP_Class is currently obsolete. It is supported for backward compatibility but its functionality is available using pragma Import with Convention = CPP.

2.39 Pragma CPP_Constructor

Syntax:

```
pragma CPP_Constructor ([Entity =>] LOCAL_NAME
  [, [External_Name =>] static_string_EXPRESSION ]
  [, [Link_Name     =>] static_string_EXPRESSION ]);
```

This pragma identifies an imported function (imported in the usual way with pragma Import) as corresponding to a C++ constructor. If External_Name and Link_Name are not specified then the Entity argument is a name that must have been previously mentioned in a pragma Import with Convention = CPP. Such name must be of one of the following forms:

* 'function' Fname 'return' T'
* 'function' Fname 'return' T'Class
* 'function' Fname (...) 'return' T'
* 'function' Fname (...) 'return' T'Class

where T is a limited record type imported from C++ with pragma Import and Convention = CPP.

The first two forms import the default constructor, used when an object of type T is created on the Ada side with no explicit constructor. The latter two forms cover all the non-default constructors of the type. See the GNAT User's Guide for details.

If no constructors are imported, it is impossible to create any objects on the Ada side and the type is implicitly declared abstract.

Pragma CPP_Constructor is intended primarily for automatic generation using an automatic binding generator tool (such as the -fdump-ada-spec GCC switch). See [Interfacing to C++], page 281, for more related information.

Note: The use of functions returning class-wide types for constructors is currently obsolete. They are supported for backward compatibility. The use of functions returning the type T leave the Ada sources more clear because the imported C++ constructors always return an object of type T; that is, they never return an object whose type is a descendant of type T.

2.40 Pragma CPP_Virtual

This pragma is now obsolete and, other than generating a warning if warnings on obsolescent features are enabled, is completely ignored. It is retained for compatibility purposes. It used to be required to ensure compatibility with C++, but is no longer required for that purpose because GNAT generates the same object layout as the G++ compiler by default.

See [Interfacing to C++], page 281, for related information.

2.41 Pragma CPP_Vtable

This pragma is now obsolete and, other than generating a warning if warnings on obsolescent features are enabled, is completely ignored. It used to be required to ensure compatibility with C++, but is no longer required for that purpose because GNAT generates the same object layout as the G++ compiler by default.

See [Interfacing to C++], page 281, for related information.

2.42 Pragma CPU

Syntax:

```
pragma CPU (EXPRESSION);
```

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.

2.43 Pragma Deadline_Floor

Syntax:

```
pragma Deadline_Floor (time_span_EXPRESSION);
```

This pragma applies only to protected types and specifies the floor deadline inherited by a task when the task enters a protected object. It is effective only when the EDF scheduling policy is used.

2.44 Pragma Debug

Syntax:

```
pragma Debug ([CONDITION, ]PROCEDURE_CALL_WITHOUT_SEMICOLON);

PROCEDURE_CALL_WITHOUT_SEMICOLON ::=
  PROCEDURE_NAME
| PROCEDURE_PREFIX ACTUAL_PARAMETER_PART
```

The procedure call argument has the syntactic form of an expression, meeting the syntactic requirements for pragmas.

If debug pragmas are not enabled or if the condition is present and evaluates to False, this pragma has no effect. If debug pragmas are enabled, the semantics of the pragma is exactly equivalent to the procedure call statement corresponding to the argument with a terminating semicolon. Pragmas are permitted in sequences of declarations, so you can use pragma `Debug` to intersperse calls to debug procedures in the middle of declarations. Debug pragmas can be enabled either by use of the command line switch '-gnata' or by use of the pragma `Check_Policy` with a first argument of `Debug`.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.45 Pragma Debug_Policy

Syntax:

```
pragma Debug_Policy (CHECK | DISABLE | IGNORE | ON | OFF);
```

This pragma is equivalent to a corresponding `Check_Policy` pragma with a first argument of `Debug`. It is retained for historical compatibility reasons.

2.46 Pragma Default_Initial_Condition

Syntax:

```
pragma Default_Initial_Condition [ (null | boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Default_Initial_Condition` in the SPARK 2014 Reference Manual, section 7.3.3.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.47 Pragma Default_Scalar_Storage_Order

Syntax:

```
pragma Default_Scalar_Storage_Order (High_Order_First | Low_Order_First);
```

Normally if no explicit `Scalar_Storage_Order` is given for a record type or array type, then the scalar storage order defaults to the ordinary default for the target. But this default may be overridden using this pragma. The pragma may appear as a configuration pragma, or locally within a package spec or declarative part. In the latter case, it applies to all subsequent types declared within that package spec or declarative part.

The following example shows the use of this pragma:

```
pragma Default_Scalar_Storage_Order (High_Order_First);
with System; use System;
package DSSO1 is
   type H1 is record
      a : Integer;
   end record;

   type L2 is record
      a : Integer;
   end record;
```

```
   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 DSSO1;
```

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.

2.52 Pragma Dispatching_Domain

Syntax:

```
pragma Dispatching_Domain (EXPRESSION);
```

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.

2.53 Pragma Effective_Reads

Syntax:

```
pragma Effective_Reads [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Effective_Reads` in the SPARK 2014 Reference Manual, section 7.1.2.

2.54 Pragma Effective_Writes

Syntax:

```
pragma Effective_Writes [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Effective_Writes` in the SPARK 2014 Reference Manual, section 7.1.2.

2.55 Pragma Elaboration_Checks

Syntax:

```
pragma Elaboration_Checks (Dynamic | Static);
```

This is a configuration pragma which specifies the elaboration model to be used during compilation. For more information on the elaboration models of GNAT, consult the chapter on elaboration order handling in the 'GNAT User's Guide'.

The pragma may appear in the following contexts:

* Configuration pragmas file
* Prior to the context clauses of a compilation unit's initial declaration

Any other placement of the pragma will result in a warning and the effects of the offending pragma will be ignored.

If the pragma argument is `Dynamic`, then the dynamic elaboration model is in effect. If the pragma argument is `Static`, then the static elaboration model is in effect.

2.56 Pragma Eliminate

Syntax:

```
pragma Eliminate (
          [  Unit_Name       => ] IDENTIFIER | SELECTED_COMPONENT ,
          [  Entity          => ] IDENTIFIER |
                                  SELECTED_COMPONENT |
                                  STRING_LITERAL
          [, Source_Location =>   SOURCE_TRACE ] );
```

```
                SOURCE_TRACE    ::= STRING_LITERAL
```

This pragma indicates that the given entity is not used in the program to be compiled and built, thus allowing the compiler to eliminate the code or data associated with the named entity. Any reference to an eliminated entity causes a compile-time or link-time error.

The pragma has the following semantics, where `U` is the unit specified by the `Unit_Name` argument and `E` is the entity specified by the `Entity` argument:

* `E` must be a subprogram that is explicitly declared either:
  * Within `U`, or
  * Within a generic package that is instantiated in `U`, or
  * As an instance of generic subprogram instantiated in `U`.

  Otherwise the pragma is ignored.
* If `E` is overloaded within `U` then, in the absence of a `Source_Location` argument, all overloadings are eliminated.
* If `E` is overloaded within `U` and only some overloadings are to be eliminated, then each overloading to be eliminated must be specified in a corresponding pragma `Eliminate` with a `Source_Location` argument identifying the line where the declaration appears, as described below.
* If `E` is declared as the result of a generic instantiation, then a `Source_Location` argument is needed, as described below.

Pragma `Eliminate` allows a program to be compiled in a system-independent manner, so that unused entities are eliminated but without needing to modify the source text. Normally the required set of `Eliminate` pragmas is constructed automatically using the `gnatelim` tool.

Any source file change that removes, splits, or adds lines may make the set of `Eliminate` pragmas invalid because their `Source_Location` argument values may get out of date.

Pragma `Eliminate` may be used where the referenced entity is a dispatching operation. In this case all the subprograms to which the given operation can dispatch are considered to be unused (are never called as a result of a direct or a dispatching call).

The string literal given for the source location specifies the line number of the declaration of the entity, using the following syntax for `SOURCE_TRACE`:

```
SOURCE_TRACE     ::= SOURCE_REFERENCE [ LBRACKET SOURCE_TRACE RBRACKET ]

LBRACKET         ::= '['
RBRACKET         ::= ']'

SOURCE_REFERENCE ::= FILE_NAME : LINE_NUMBER

LINE_NUMBER      ::= DIGIT {DIGIT}
```

Spaces around the colon in a `SOURCE_REFERENCE` are optional.

The source trace that is given as the `Source_Location` must obey the following rules (or else the pragma is ignored), where `U` is the unit `U` specified by the `Unit_Name` argument and `E` is the subprogram specified by the `Entity` argument:

* `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]");
```

2.57 Pragma Enable_Atomic_Synchronization

Syntax:

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

Reenables atomic synchronization; see `pragma Disable_Atomic_Synchronization` for details.

2.58 Pragma Exceptional_Cases

Syntax:

```
pragma Exceptional_Cases (EXCEPTIONAL_CASE_LIST);

EXCEPTIONAL_CASE_LIST ::= EXCEPTIONAL_CASE {, EXCEPTIONAL_CASE}
EXCEPTIONAL_CASE      ::= exception_choice {'|' exception_choice} => CONSEQUENCE
CONSEQUENCE           ::= Boolean_expression
```

For the semantics of this aspect, see the SPARK 2014 Reference Manual, section 6.1.9.

2.59 Pragma Exit_Cases

Syntax:

```
pragma Exit_Cases (EXIT_CASE_LIST);

EXIT_CASE_LIST ::= EXIT_CASE {, EXIT_CASE}
EXIT_CASE      ::= GUARD => EXIT_KIND
EXIT_KIND      ::= Normal_Return
                 | Exception_Raised
                 | (Exception_Raised => exception_name)
                 | Program_Exit
GUARD          ::= Boolean_expression
```

For the semantics of this aspect, see the SPARK 2014 Reference Manual, section 6.1.10.

2.60 Pragma Export_Function

Syntax:

```
pragma Export_Function (
     [Internal         =>] LOCAL_NAME
  [, [External         =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types  =>] PARAMETER_TYPES]
  [, [Result_Type      =>] result_SUBTYPE_MARK]
  [, [Mechanism        =>] MECHANISM]
  [, [Result_Mechanism =>] MECHANISM_NAME]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
| ""
```

```
PARAMETER_TYPES ::=
  null
| TYPE_DESIGNATOR {, TYPE_DESIGNATOR}

TYPE_DESIGNATOR ::=
  subtype_NAME
| subtype_Name ' Access

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::= Value | Reference
```

Use this pragma to make a function externally callable and optionally provide information on mechanisms to be used for passing parameter and result values. We recommend, for the purposes of improving portability, this pragma always be used in conjunction with a separate pragma `Export`, which must precede the pragma `Export_Function`. GNAT does not require a separate pragma `Export`, but if none is present, `Convention Ada` is assumed, which is usually not what is wanted, so it is usually appropriate to use this pragma in conjunction with a `Export` or `Convention` pragma that specifies the desired foreign convention. Pragma `Export_Function` (and `Export`, if present) must appear in the same declarative region as the function to which they apply.

The `internal_name` must uniquely designate the function to which the pragma applies. If more than one function name exists of this name in the declarative part you must use the `Parameter_Types` and `Result_Type` parameters to achieve the required unique designation. The *subtype_marks* in these parameters must exactly match the subtypes in the corresponding function specification, using positional notation to match parameters with subtype marks. The form with an `'Access` attribute can be used to match an anonymous access parameter.

Special treatment is given if the EXTERNAL is an explicit null string or a static string expressions that evaluates to the null string. In this case, no external name is generated. This form still allows the specification of parameter mechanisms.

2.61 Pragma Export_Object

Syntax:

```
pragma Export_Object (
      [Internal =>] LOCAL_NAME
   [, [External =>] EXTERNAL_SYMBOL]
   [, [Size     =>] EXTERNAL_SYMBOL]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
```

```
| static_string_EXPRESSION
```

This pragma designates an object as exported, and apart from the extended rules for external symbols, is identical in effect to the use of the normal `Export` pragma applied to an object. You may use a separate Export pragma (and you probably should from the point of view of portability), but it is not required. `Size` is syntax checked, but otherwise ignored by GNAT.

2.62 Pragma Export_Procedure

Syntax:

```
pragma Export_Procedure (
     [Internal        =>] LOCAL_NAME
  [, [External        =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types =>] PARAMETER_TYPES]
  [, [Mechanism       =>] MECHANISM]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
| ""

PARAMETER_TYPES ::=
  null
| TYPE_DESIGNATOR {, TYPE_DESIGNATOR}

TYPE_DESIGNATOR ::=
  subtype_NAME
| subtype_Name ' Access

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::= Value | Reference
```

This pragma is identical to `Export_Function` except that it applies to a procedure rather than a function and the parameters `Result_Type` and `Result_Mechanism` are not permitted. GNAT does not require a separate pragma `Export`, but if none is present, `Convention Ada` is assumed, which is usually not what is wanted, so it is usually appropriate to use this pragma in conjunction with a `Export` or `Convention` pragma that specifies the desired foreign convention.

Special treatment is given if the EXTERNAL is an explicit null string or a static string expressions that evaluates to the null string. In this case, no external name is generated. This form still allows the specification of parameter mechanisms.

2.63 Pragma Export_Valued_Procedure

Syntax:

```
pragma Export_Valued_Procedure (
     [Internal        =>] LOCAL_NAME
  [, [External        =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types =>] PARAMETER_TYPES]
  [, [Mechanism       =>] MECHANISM]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
| ""

PARAMETER_TYPES ::=
  null
| TYPE_DESIGNATOR {, TYPE_DESIGNATOR}

TYPE_DESIGNATOR ::=
  subtype_NAME
| subtype_Name ' Access

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::= Value | Reference
```

This pragma is identical to `Export_Procedure` except that the first parameter of `LOCAL_NAME`, which must be present, must be of mode `out`, and externally the subprogram is treated as a function with this parameter as the result of the function. GNAT provides for this capability to allow the use of `out` and `in out` parameters in interfacing to external functions (which are not permitted in Ada functions). GNAT does not require a separate pragma `Export`, but if none is present, `Convention Ada` is assumed, which is almost certainly not what is wanted since the whole point of this pragma is to interface with foreign language functions, so it is usually appropriate to use this pragma in conjunction with a `Export` or `Convention` pragma that specifies the desired foreign convention.

Special treatment is given if the EXTERNAL is an explicit null string or a static string expressions that evaluates to the null string. In this case, no external name is generated. This form still allows the specification of parameter mechanisms.

2.64 Pragma Extend_System

Syntax:

```
pragma Extend_System ([Name =>] IDENTIFIER);
```

This pragma is used to provide backwards compatibility with other implementations that extend the facilities of package `System`. In GNAT, `System` contains only the definitions that are present in the Ada RM. However, other implementations, notably the DEC Ada 83 implementation, provide many extensions to package `System`.

For each such implementation accommodated by this pragma, GNAT provides a package `Aux_xxx`, e.g., `Aux_DEC` for the DEC Ada 83 implementation, which provides the required additional definitions. You can use this package in two ways. You can `with` it in the normal way and access entities either by selection or using a `use` clause. In this case no special processing is required.

However, if existing code contains references such as `System.xxx` where 'xxx' is an entity in the extended definitions provided in package `System`, you may use this pragma to extend visibility in `System` in a non-standard way that provides greater compatibility with the existing code. Pragma `Extend_System` is a configuration pragma whose single argument is the name of the package containing the extended definition (e.g., `Aux_DEC` for the DEC Ada case). A unit compiled under control of this pragma will be processed using special visibility processing that looks in package `System.Aux_xxx` where `Aux_xxx` is the pragma argument for any entity referenced in package `System`, but not found in package `System`.

You can use this pragma either to access a predefined `System` extension supplied with the compiler, for example `Aux_DEC` or you can construct your own extension unit following the above definition. Note that such a package is a child of `System` and thus is considered part of the implementation. To compile it you will have to use the '-gnatg' switch for compiling System units, as explained in the GNAT User's Guide.

2.65 Pragma Extensions_Allowed

Syntax:

```
pragma Extensions_Allowed (On | Off | All_Extensions);
```

This configuration pragma enables (via the "On" or "All_Extensions" argument) or disables (via the "Off" argument) the implementation extension mode; the pragma takes precedence over the `-gnatX` and `-gnatX0` command switches.

If an argument of `"On"` is specified, the latest version of the Ada language is implemented (currently Ada 2022) and, in addition, a curated set of GNAT specific extensions are recognized. (See the list here [here], page 330)

An argument of `"All_Extensions"` has the same effect except that some extra experimental extensions are enabled (See the list here [here], page 341)

2.66 Pragma Extensions_Visible

Syntax:

```
pragma Extensions_Visible [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Extensions_Visible` in the SPARK 2014 Reference Manual, section 6.1.7.

2.67 Pragma External

Syntax:

```
pragma External (
  [   Convention    =>] convention_IDENTIFIER,
  [   Entity        =>] LOCAL_NAME
  [, [External_Name =>] static_string_EXPRESSION ]
  [, [Link_Name     =>] static_string_EXPRESSION ]);
```

This pragma is identical in syntax and semantics to pragma `Export` as defined in the Ada Reference Manual. It is provided for compatibility with some Ada 83 compilers that used this pragma for exactly the same purposes as pragma `Export` before the latter was standardized.

2.68 Pragma External_Name_Casing

Syntax:

```
pragma External_Name_Casing (
  Uppercase | Lowercase
  [, Uppercase | Lowercase | As_Is]);
```

This pragma provides control over the casing of external names associated with Import and Export pragmas. There are two cases to consider:

* Implicit external names

  Implicit external names are derived from identifiers. The most common case arises when a standard Ada Import or Export pragma is used with only two arguments, as in:

  ```
  pragma Import (C, C_Routine);
  ```

  Since Ada is a case-insensitive language, the spelling of the identifier in the Ada source program does not provide any information on the desired casing of the external name, and so a convention is needed. In GNAT the default treatment is that such names are converted to all lower case letters. This corresponds to the normal C style in many environments. The first argument of pragma `External_Name_Casing` can be used to control this treatment. If `Uppercase` is specified, then the name will be forced to all uppercase letters. If `Lowercase` is specified, then the normal default of all lower case letters will be used.

  This same implicit treatment is also used in the case of extended DEC Ada 83 compatible Import and Export pragmas where an external name is explicitly specified using an identifier rather than a string.

* Explicit external names

  Explicit external names are given as string literals. The most common case arises when a standard Ada Import or Export pragma is used with three arguments, as in:

  ```
  pragma Import (C, C_Routine, "C_routine");
  ```

  In this case, the string literal normally provides the exact casing required for the external name. The second argument of pragma `External_Name_Casing` may be used to modify this behavior. If `Uppercase` is specified, then the name will be forced to all uppercase letters. If `Lowercase` is specified, then the name will be forced to all lowercase letters. A specification of `As_Is` provides the normal default behavior in which the casing is taken from the string provided.

This pragma may appear anywhere that a pragma is valid. In particular, it can be used as a configuration pragma in the `gnat.adc` file, in which case it applies to all subsequent compilations, or it can be used as a program unit pragma, in which case it only applies to the current unit, or it can be used more locally to control individual Import/Export pragmas.

It was primarily intended for use with OpenVMS systems, where many compilers convert all symbols to upper case by default. For interfacing to such compilers (e.g., the DEC C compiler), it may be convenient to use the pragma:

```
pragma External_Name_Casing (Uppercase, Uppercase);
```

to enforce the upper casing of all external symbols.

2.69 Pragma Fast_Math

Syntax:

```
pragma Fast_Math;
```

This is a configuration pragma which activates a mode in which speed is considered more important for floating-point operations than absolutely accurate adherence to the requirements of the standard. Currently the following operations are affected:

'Complex Multiplication'
: The normal simple formula for complex multiplication can result in intermediate overflows for numbers near the end of the range. The Ada standard requires that this situation be detected and corrected by scaling, but in Fast_Math mode such cases will simply result in overflow. Note that to take advantage of this you must instantiate your own version of `Ada.Numerics.Generic_Complex_Types` under control of the pragma, rather than use the preinstantiated versions.

2.70 Pragma Favor_Top_Level

Syntax:

```
pragma Favor_Top_Level (type_LOCAL_NAME);
```

The argument of pragma `Favor_Top_Level` must be a named access-to-subprogram type. This pragma is an efficiency hint to the compiler, regarding the use of `'Access` or `'Unrestricted_Access` on nested (non-library-level) subprograms. The pragma means that nested subprograms are not used with this type, or are rare, so that the generated code should be efficient in the top-level case. When this pragma is used, dynamically generated trampolines may be used on some targets for nested subprograms. See restriction `No_Implicit_Dynamic_Code`.

2.71 Pragma Finalize_Storage_Only

Syntax:

```
pragma Finalize_Storage_Only (first_subtype_LOCAL_NAME);
```

The argument of pragma `Finalize_Storage_Only` must denote a local type which is derived from `Ada.Finalization.Controlled` or `Limited_Controlled`. The pragma suppresses the call to `Finalize` for declared library-level objects of the argument type. This is mostly useful for types where finalization is only used to deal with storage reclamation since in most environments it is not necessary to reclaim memory just before terminating execution, hence the name. Note that this pragma does not suppress Finalize calls for library-level heap-allocated objects (see pragma `No_Heap_Finalization`).

2.72 Pragma Float_Representation

Syntax:

```
pragma Float_Representation (FLOAT_REP[, float_type_LOCAL_NAME]);

FLOAT_REP ::= VAX_Float | IEEE_Float
```

In the one argument form, this pragma is a configuration pragma which allows control over the internal representation chosen for the predefined floating point types declared in the packages `Standard` and `System`. This pragma is only provided for compatibility and has no effect.

The two argument form specifies the representation to be used for the specified floating-point type. The argument must be `IEEE_Float` to specify the use of IEEE format, as follows:

* For a digits value of 6, 32-bit IEEE short format will be used.
* For a digits value of 15, 64-bit IEEE long format will be used.
* No other value of digits is permitted.

2.73 Pragma Ghost

Syntax:

```
pragma Ghost [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Ghost` in the SPARK 2014 Reference Manual, section 6.9.

2.74 Pragma Global

Syntax:

```
pragma Global (GLOBAL_SPECIFICATION);

GLOBAL_SPECIFICATION ::=
     null
  | (GLOBAL_LIST)
  | (MODED_GLOBAL_LIST {, MODED_GLOBAL_LIST})

MODED_GLOBAL_LIST ::= MODE_SELECTOR => GLOBAL_LIST

MODE_SELECTOR ::= In_Out | Input | Output | Proof_In
GLOBAL_LIST   ::= GLOBAL_ITEM | (GLOBAL_ITEM {, GLOBAL_ITEM})
```

```
GLOBAL_ITEM   ::= NAME
```

For the semantics of this pragma, see the entry for aspect `Global` in the SPARK 2014 Reference Manual, section 6.1.4.

2.75 Pragma Ident

Syntax:

```
pragma Ident (static_string_EXPRESSION);
```

This pragma is identical in effect to pragma `Comment`. It is provided for compatibility with other Ada compilers providing this pragma.

2.76 Pragma Ignore_Pragma

Syntax:

```
pragma Ignore_Pragma (pragma_IDENTIFIER);
```

This is a configuration pragma that takes a single argument that is a simple identifier. Any subsequent use of a pragma whose pragma identifier matches this argument will be silently ignored. Any preceding use of a pragma whose pragma identifier matches this argument will be parsed and then ignored. This may be useful when legacy code or code intended for compilation with some other compiler contains pragmas that match the name, but not the exact implementation, of a GNAT pragma. The use of this pragma allows such pragmas to be ignored, which may be useful in CodePeer mode, or during porting of legacy code.

2.77 Pragma Implementation_Defined

Syntax:

```
pragma Implementation_Defined (local_NAME);
```

This pragma marks a previously declared entity as implementation-defined. For an overloaded entity, applies to the most recent homonym.

```
pragma Implementation_Defined;
```

The form with no arguments appears anywhere within a scope, most typically a package spec, and indicates that all entities that are defined within the package spec are Implementation_Defined.

This pragma is used within the GNAT runtime library to identify implementation-defined entities introduced in language-defined units, for the purpose of implementing the No_Implementation_Identifiers restriction.

2.78 Pragma Implemented

Syntax:

```
pragma Implemented (procedure_LOCAL_NAME, implementation_kind);

implementation_kind ::= By_Entry | By_Protected_Procedure | By_Any
```

This is an Ada 2012 representation pragma which applies to protected, task and synchronized interface primitives. The use of pragma Implemented provides a way to impose a static requirement on the overriding operation by adhering to one of the three implementation kinds: entry, protected procedure or any of the above. This pragma is available in all earlier versions of Ada as an implementation-defined pragma.

```
type Synch_Iface is synchronized interface;
procedure Prim_Op (Obj : in out Iface) is abstract;
pragma Implemented (Prim_Op, By_Protected_Procedure);

protected type Prot_1 is new Synch_Iface with
   procedure Prim_Op;  --  Legal
end Prot_1;

protected type Prot_2 is new Synch_Iface with
   entry Prim_Op;      --  Illegal
end Prot_2;

task type Task_Typ is new Synch_Iface with
   entry Prim_Op;      --  Illegal
end Task_Typ;
```

When applied to the procedure_or_entry_NAME of a requeue statement, pragma Implemented determines the runtime behavior of the requeue. Implementation kind By_Entry guarantees that the action of requeueing will proceed from an entry to another entry. Implementation kind By_Protected_Procedure transforms the requeue into a dispatching call, thus eliminating the chance of blocking. Kind By_Any shares the behavior of By_Entry and By_Protected_Procedure depending on the target's overriding subprogram kind.

2.79 Pragma Implicit_Packing

Syntax:

```
pragma Implicit_Packing;
```

This is a configuration pragma that requests implicit packing for packed arrays for which a size clause is given but no explicit pragma Pack or specification of Component_Size is present. It also applies to records where no record representation clause is present. Consider this example:

```
type R is array (0 .. 7) of Boolean;
for R'Size use 8;
```

In accordance with the recommendation in the RM (RM 13.3(53)), a Size clause does not change the layout of a composite object. So the Size clause in the above example is normally rejected, since the default layout of the array uses 8-bit components, and thus the array requires a minimum of 64 bits.

If this declaration is compiled in a region of code covered by an occurrence of the configuration pragma Implicit_Packing, then the Size clause in this and similar examples will cause implicit packing and thus be accepted. For this implicit packing to occur, the type in question must be an array of small components whose size is known at compile time, and the Size clause must specify the exact size that corresponds to the number of elements in the array multiplied by the size in bits of the component type (both single and multi-dimensioned arrays can be controlled with this pragma).

Similarly, the following example shows the use in the record case

```
type r is record
   a, b, c, d, e, f, g, h : boolean;
   chr                    : character;
end record;
for r'size use 16;
```

Without a pragma Pack, each Boolean field requires 8 bits, so the minimum size is 72 bits, but with a pragma Pack, 16 bits would be sufficient. The use of pragma Implicit_Packing allows this record declaration to compile without an explicit pragma Pack.

2.80 Pragma Import_Function

Syntax:

```
pragma Import_Function (
     [Internal         =>] LOCAL_NAME,
  [, [External         =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types  =>] PARAMETER_TYPES]
  [, [Result_Type      =>] SUBTYPE_MARK]
  [, [Mechanism        =>] MECHANISM]
  [, [Result_Mechanism =>] MECHANISM_NAME]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| TYPE_DESIGNATOR {, TYPE_DESIGNATOR}

TYPE_DESIGNATOR ::=
  subtype_NAME
| subtype_Name ' Access

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::=
  Value
| Reference
```

This pragma is used in conjunction with a pragma `Import` to specify additional information for an imported function. The pragma `Import` (or equivalent pragma `Interface`) must precede the `Import_Function` pragma and both must appear in the same declarative part as the function specification.

The `Internal` argument must uniquely designate the function to which the pragma applies. If more than one function name exists of this name in the declarative part you must use the `Parameter_Types` and `Result_Type` parameters to achieve the required unique designation. Subtype marks in these parameters must exactly match the subtypes in the corresponding function specification, using positional notation to match parameters with subtype marks. The form with an `'Access` attribute can be used to match an anonymous access parameter.

You may optionally use the `Mechanism` and `Result_Mechanism` parameters to specify passing mechanisms for the parameters and result. If you specify a single mechanism name, it applies to all parameters. Otherwise you may specify a mechanism on a parameter by parameter basis using either positional or named notation. If the mechanism is not specified, the default mechanism is used.

2.81 Pragma Import_Object

Syntax:

```
pragma Import_Object (
     [Internal =>] LOCAL_NAME
  [, [External =>] EXTERNAL_SYMBOL]
  [, [Size     =>] EXTERNAL_SYMBOL]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma designates an object as imported, and apart from the extended rules for external symbols, is identical in effect to the use of the normal `Import` pragma applied to an object. Unlike the subprogram case, you need not use a separate `Import` pragma, although you may do so (and probably should do so from a portability point of view). `size` is syntax checked, but otherwise ignored by GNAT.

2.82 Pragma Import_Procedure

Syntax:

```
pragma Import_Procedure (
     [Internal        =>] LOCAL_NAME
  [, [External        =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types =>] PARAMETER_TYPES]
  [, [Mechanism       =>] MECHANISM]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
```

```
| TYPE_DESIGNATOR {, TYPE_DESIGNATOR}

TYPE_DESIGNATOR ::=
  subtype_NAME
| subtype_Name ' Access

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::= Value | Reference
```

This pragma is identical to `Import_Function` except that it applies to a procedure rather than a function and the parameters `Result_Type` and `Result_Mechanism` are not permitted.

2.83 Pragma Import_Valued_Procedure

Syntax:

```
pragma Import_Valued_Procedure (
     [Internal        =>] LOCAL_NAME
  [, [External        =>] EXTERNAL_SYMBOL]
  [, [Parameter_Types =>] PARAMETER_TYPES]
  [, [Mechanism       =>] MECHANISM]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION

PARAMETER_TYPES ::=
  null
| TYPE_DESIGNATOR {, TYPE_DESIGNATOR}

TYPE_DESIGNATOR ::=
  subtype_NAME
| subtype_Name ' Access

MECHANISM ::=
  MECHANISM_NAME
| (MECHANISM_ASSOCIATION {, MECHANISM_ASSOCIATION})

MECHANISM_ASSOCIATION ::=
  [formal_parameter_NAME =>] MECHANISM_NAME

MECHANISM_NAME ::= Value | Reference
```

This pragma is identical to `Import_Procedure` except that the first parameter of `LOCAL_NAME`, which must be present, must be of mode `out`, and externally the subprogram is treated as a function with this parameter as the result of the function. The purpose of this capability is to allow the use of `out` and `in out` parameters in interfacing to external functions (which are not permitted in Ada functions). You may optionally use the `Mechanism` parameters to specify passing mechanisms for the parameters. If you specify a single mechanism name, it applies to all parameters. Otherwise you may specify a mechanism on a parameter by parameter basis using either positional or named notation. If the mechanism is not specified, the default mechanism is used.

Note that it is important to use this pragma in conjunction with a separate pragma Import that specifies the desired convention, since otherwise the default convention is Ada, which is almost certainly not what is required.

2.84 Pragma Independent

Syntax:

```
pragma Independent (component_LOCAL_NAME);
```

This pragma is standard in Ada 2012 mode (which also provides an aspect of the same name). It is also available as an implementation-defined pragma in all earlier versions. It specifies that the designated object or all objects of the designated type must be independently addressable. This means that separate tasks can safely manipulate such objects. For example, if two components of a record are independent, then two separate tasks may access these two components. This may place constraints on the representation of the object (for instance prohibiting tight packing).

2.85 Pragma Independent_Components

Syntax:

```
pragma Independent_Components (Local_NAME);
```

This pragma is standard in Ada 2012 mode (which also provides an aspect of the same name). It is also available as an implementation-defined pragma in all earlier versions. It specifies that the components of the designated object, or the components of each object of the designated type, must be independently addressable. This means that separate tasks can safely manipulate separate components in the composite object. This may place constraints on the representation of the object (for instance prohibiting tight packing).

2.86 Pragma Initial_Condition

Syntax:

```
pragma Initial_Condition (boolean_EXPRESSION);
```

For the semantics of this pragma, see the entry for aspect `Initial_Condition` in the SPARK 2014 Reference Manual, section 7.1.6.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.87 Pragma Initialize_Scalars

Syntax:

```
pragma Initialize_Scalars
  [ ( TYPE_VALUE_PAIR {, TYPE_VALUE_PAIR} ) ];

TYPE_VALUE_PAIR ::=
  SCALAR_TYPE => static_EXPRESSION

SCALAR_TYPE :=
  Short_Float
| Float
| Long_Float
| Long_Long_Flat
| Signed_8
| Signed_16
| Signed_32
| Signed_64
| Unsigned_8
| Unsigned_16
| Unsigned_32
| Unsigned_64
```

This pragma is similar to `Normalize_Scalars` conceptually but has two important differences.

First, there is no requirement for the pragma to be used uniformly in all units of a partition. In particular, it is fine to use this just for some or all of the application units of a partition, without needing to recompile the run-time library. In the case where some units are compiled with the pragma, and some without, then a declaration of a variable where the type is defined in package Standard or is locally declared will always be subject to initialization, as will any declaration of a scalar variable. For composite variables, whether the variable is initialized may also depend on whether the package in which the type of the variable is declared is compiled with the pragma.

The other important difference is that the programmer can control the value used for initializing scalar objects. This effect can be achieved in several different ways:

* At compile time, the programmer can specify the invalid value for a particular family of scalar types using the optional arguments of the pragma.

  The compile-time approach is intended to optimize the generated code for the pragma, by possibly using fast operations such as `memset`. Note that such optimizations require using values where the bytes all have the same binary representation.

* At bind time, the programmer has several options:
  * Initialization with invalid values (similar to Normalize_Scalars, though for Initialize_Scalars it is not always possible to determine the invalid values in complex cases like signed component fields with nonstandard sizes).
  * Initialization with high values.
  * Initialization with low values.

  * Initialization with a specific bit pattern.

  See the GNAT User's Guide for binder options for specifying these cases.

  The bind-time approach is intended to provide fast turnaround for testing with different values, without having to recompile the program.

* At execution time, the programmer can specify the invalid values using an environment variable. See the GNAT User's Guide for details.

  The execution-time approach is intended to provide fast turnaround for testing with different values, without having to recompile and rebind the program.

Note that pragma `Initialize_Scalars` is particularly useful in conjunction with the enhanced validity checking that is now provided in GNAT, which checks for invalid values under more conditions. Using this feature (see description of the '-gnatV' flag in the GNAT User's Guide) in conjunction with pragma `Initialize_Scalars` provides a powerful new tool to assist in the detection of problems caused by uninitialized variables.

Note: the use of `Initialize_Scalars` has a fairly extensive effect on the generated code. This may cause your code to be substantially larger. It may also cause an increase in the amount of stack required, so it is probably a good idea to turn on stack checking (see description of stack checking in the GNAT User's Guide) when using this pragma.

2.88 Pragma Initializes

Syntax:

```
pragma Initializes (INITIALIZATION_LIST);

INITIALIZATION_LIST ::=
     null
  | (INITIALIZATION_ITEM {, INITIALIZATION_ITEM})

INITIALIZATION_ITEM ::= name [=> INPUT_LIST]

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

INPUT ::= name
```

For the semantics of this pragma, see the entry for aspect `Initializes` in the SPARK 2014 Reference Manual, section 7.1.5.

2.89 Pragma Inline_Always

Syntax:

```
pragma Inline_Always (NAME [, NAME]);
```

Similar to pragma `Inline` except that inlining is unconditional. Inline_Always instructs the compiler to inline every direct call to the subprogram or else to emit a compilation error, independently of any option, in particular '-gnatn' or '-gnatN' or the optimization level. It is an error to take the address or access of `NAME`. It is also an error to apply this pragma to a primitive operation of a tagged type. Thanks to such restrictions, the compiler is allowed to remove the out-of-line body of `NAME`.

2.90 Pragma Inline_Generic

Syntax:

```
pragma Inline_Generic (GNAME {, GNAME});

GNAME ::= generic_unit_NAME | generic_instance_NAME
```

This pragma is provided for compatibility with Dec Ada 83. It has no effect in GNAT (which always inlines generics), other than to check that the given names are all names of generic units or generic instances.

2.91 Pragma Interface

Syntax:

```
pragma Interface (
     [Convention    =>] convention_identifier,
     [Entity        =>] local_NAME
  [, [External_Name =>] static_string_expression]
  [, [Link_Name     =>] static_string_expression]);
```

This pragma is identical in syntax and semantics to the standard Ada pragma `Import`. It is provided for compatibility with Ada 83. The definition is upwards compatible both with pragma `Interface` as defined in the Ada 83 Reference Manual, and also with some extended implementations of this pragma in certain Ada 83 implementations. The only difference between pragma `Interface` and pragma `Import` is that there is special circuitry to allow both pragmas to appear for the same subprogram entity (normally it is illegal to have multiple `Import` pragmas). This is useful in maintaining Ada 83/Ada 95 compatibility and is compatible with other Ada 83 compilers.

2.92 Pragma Interface_Name

Syntax:

```
pragma Interface_Name (
     [Entity        =>] LOCAL_NAME
  [, [External_Name =>] static_string_EXPRESSION]
  [, [Link_Name     =>] static_string_EXPRESSION]);
```

This pragma provides an alternative way of specifying the interface name for an interfaced subprogram, and is provided for compatibility with Ada 83 compilers that use the pragma for this purpose. You must provide at least one of `External_Name` or `Link_Name`.

2.93 Pragma Interrupt_Handler

Syntax:

```
pragma Interrupt_Handler (procedure_LOCAL_NAME);
```

This program unit pragma is supported for parameterless protected procedures as described in Annex C of the Ada Reference Manual.

2.94 Pragma Interrupt_State

Syntax:

```
pragma Interrupt_State
 ([Name  =>] value,
  [State =>] SYSTEM | RUNTIME | USER);
```

Normally certain interrupts are reserved to the implementation. Any attempt to attach an interrupt causes Program_Error to be raised, as described in RM C.3.2(22). A typical example is the `SIGINT` interrupt used in many systems for an `Ctrl-C` interrupt. Normally this interrupt is reserved to the implementation, so that `Ctrl-C` can be used to interrupt execution. Additionally, signals such as `SIGSEGV`, `SIGABRT`, `SIGFPE` and `SIGILL` are often mapped to specific Ada exceptions, or used to implement run-time functions such as the `abort` statement and stack overflow checking.

Pragma `Interrupt_State` provides a general mechanism for overriding such uses of interrupts. It subsumes the functionality of pragma `Unreserve_All_Interrupts`. Pragma `Interrupt_State` is not available on Windows. On all other platforms than VxWorks, it applies to signals; on VxWorks, it applies to vectored hardware interrupts and may be used to mark interrupts required by the board support package as reserved.

Interrupts can be in one of three states:

* System

  The interrupt is reserved (no Ada handler can be installed), and the Ada run-time may not install a handler. As a result you are guaranteed standard system default action if this interrupt is raised. This also allows installing a low level handler via C APIs such as sigaction(), outside of Ada control.

* Runtime

  The interrupt is reserved (no Ada handler can be installed). The run time is allowed to install a handler for internal control purposes, but is not required to do so.

* User

  The interrupt is unreserved. The user may install an Ada handler via Ada.Interrupts and pragma Interrupt_Handler or Attach_Handler to provide some other action.

These states are the allowed values of the `State` parameter of the pragma. The `Name` parameter is a value of the type `Ada.Interrupts.Interrupt_ID`. Typically, it is a name declared in `Ada.Interrupts.Names`.

This is a configuration pragma, and the binder will check that there are no inconsistencies between different units in a partition in how a given interrupt is specified. It may appear anywhere a pragma is legal.

The effect is to move the interrupt to the specified state.

By declaring interrupts to be SYSTEM, you guarantee the standard system action, such as a core dump.

By declaring interrupts to be USER, you guarantee that you can install a handler.

Note that certain signals on many operating systems cannot be caught and handled by applications. In such cases, the pragma is ignored. See the operating system documentation, or the value of the array `Reserved` declared in the spec of package `System.OS_Interface`.

Overriding the default state of signals used by the Ada runtime may interfere with an application's runtime behavior in the cases of the synchronous signals, and in the case of the signal used to implement the `abort` statement.

2.95 Pragma Interrupts_System_By_Default

Syntax:

```
pragma Interrupts_System_By_Default;
```

Default all interrupts to the System state as defined above in pragma `Interrupt_State`. This is a configuration pragma.

2.96 Pragma Invariant

Syntax:

```
pragma Invariant
  ([Entity =>]    private_type_LOCAL_NAME,
   [Check  =>]    EXPRESSION
   [,[Message =>] String_Expression]);
```

This pragma provides exactly the same capabilities as the Type_Invariant aspect defined in AI05-0146-1, and in the Ada 2012 Reference Manual. The Type_Invariant aspect is fully implemented in Ada 2012 mode, but since it requires the use of the aspect syntax, which is not available except in 2012 mode, it is not possible to use the Type_Invariant aspect in earlier versions of Ada. However the Invariant pragma may be used in any version of Ada. Also note that the aspect Invariant is a synonym in GNAT for the aspect Type_Invariant, but there is no pragma Type_Invariant.

The pragma must appear within the visible part of the package specification, after the type to which its Entity argument appears. As with the Invariant aspect, the Check expression is not analyzed until the end of the visible part of the package, so it may contain forward references. The Message argument, if present, provides the exception message used if the invariant is violated. If no Message parameter is provided, a default message that identifies the line on which the pragma appears is used.

It is permissible to have multiple Invariants for the same type entity, in which case they are and'ed together. It is permissible to use this pragma in Ada 2012 mode, but you cannot have both an invariant aspect and an invariant pragma for the same entity.

For further details on the use of this pragma, see the Ada 2012 documentation of the Type_Invariant aspect.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.97 Pragma Keep_Names

Syntax:

```
pragma Keep_Names ([On =>] enumeration_first_subtype_LOCAL_NAME);
```

The LOCAL_NAME argument must refer to an enumeration first subtype in the current declarative part. The effect is to retain the enumeration literal names for use by Image and Value even if a global Discard_Names pragma applies. This is useful when you want to generally suppress enumeration literal names and for example you therefore use a Discard_Names pragma in the gnat.adc file, but you want to retain the names for specific enumeration types.

2.98 Pragma License

Syntax:

```
pragma License (Unrestricted | GPL | Modified_GPL | Restricted);
```

This pragma is provided to allow automated checking for appropriate license conditions with respect to the standard and modified GPL. A pragma License, which is a configuration pragma that typically appears at the start of a source file or in a separate gnat.adc file, specifies the licensing conditions of a unit as follows:

* Unrestricted This is used for a unit that can be freely used with no license restrictions. Examples of such units are public domain units, and units from the Ada Reference Manual.
* GPL This is used for a unit that is licensed under the unmodified GPL, and which therefore cannot be withed by a restricted unit.
* Modified_GPL This is used for a unit licensed under the GNAT modified GPL that includes a special exception paragraph that specifically permits the inclusion of the unit in programs without requiring the entire program to be released under the GPL.
* Restricted This is used for a unit that is restricted in that it is not permitted to depend on units that are licensed under the GPL. Typical examples are proprietary code that is to be released under more restrictive license conditions. Note that restricted units are permitted to with units which are licensed under the modified GPL (this is the whole point of the modified GPL).

Normally a unit with no License pragma is considered to have an unknown license, and no checking is done. However, standard GNAT headers are recognized, and license information is derived from them as follows.

A GNAT license header starts with a line containing 78 hyphens. The following comment text is searched for the appearance of any of the following strings.

If the string 'GNU General Public License' is found, then the unit is assumed to have GPL license, unless the string 'As a special exception' follows, in which case the license is assumed to be modified GPL.

If one of the strings 'This specification is adapted from the Ada Semantic Interface' or 'This specification is derived from the Ada Reference Manual' is found then the unit is assumed to be unrestricted.

These default actions means that a program with a restricted license pragma will automatically get warnings if a GPL unit is inappropriately withed. For example, the program:

```
with Sem_Ch3;
with GNAT.Sockets;
procedure Secret_Stuff is
```

```
  ...
end Secret_Stuff
```

if compiled with pragma `License` (`Restricted`) in a `gnat.adc` file will generate the warning:

```
1.  with Sem_Ch3;
        |
   >>> license of withed unit "Sem_Ch3" is incompatible

2.  with GNAT.Sockets;
3.  procedure Secret_Stuff is
```

Here we get a warning on `Sem_Ch3` since it is part of the GNAT compiler and is licensed under the GPL, but no warning for `GNAT.Sockets` which is part of the GNAT run time, and is therefore licensed under the modified GPL.

2.99 Pragma Link_With

Syntax:

```
pragma Link_With (static_string_EXPRESSION {,static_string_EXPRESSION});
```

This pragma is provided for compatibility with certain Ada 83 compilers. It has exactly the same effect as pragma `Linker_Options` except that spaces occurring within one of the string expressions are treated as separators. For example, in the following case:

```
pragma Link_With ("-labc -ldef");
```

results in passing the strings `-labc` and `-ldef` as two separate arguments to the linker. In addition pragma Link_With allows multiple arguments, with the same effect as successive pragmas.

2.100 Pragma Linker_Alias

Syntax:

```
pragma Linker_Alias (
  [Entity =>] LOCAL_NAME,
  [Target =>] static_string_EXPRESSION);
```

`LOCAL_NAME` must refer to an object that is declared at the library level. This pragma establishes the given entity as a linker alias for the given target. It is equivalent to `__attribute__((alias))` in GNU C and causes `LOCAL_NAME` to be emitted as an alias for the symbol `static_string_EXPRESSION` in the object file, that is to say no space is reserved for `LOCAL_NAME` by the assembler and it will be resolved to the same address as `static_string_EXPRESSION` by the linker.

The actual linker name for the target must be used (e.g., the fully encoded name with qualification in Ada, or the mangled name in C++), or it must be declared using the C convention with `pragma Import` or `pragma Export`.

Not all target machines support this pragma. On some of them it is accepted only if `pragma Weak_External` has been applied to `LOCAL_NAME`.

```
--  Example of the use of pragma Linker_Alias

package p is
```

```
  i : Integer := 1;
  pragma Export (C, i);

  new_name_for_i : Integer;
  pragma Linker_Alias (new_name_for_i, "i");
end p;
```

2.101 Pragma Linker_Constructor

Syntax:

```
pragma Linker_Constructor (procedure_LOCAL_NAME);
```

`procedure_LOCAL_NAME` must refer to a parameterless procedure that is declared at the library level. A procedure to which this pragma is applied will be treated as an initialization routine by the linker. It is equivalent to `__attribute__((constructor))` in GNU C and causes `procedure_LOCAL_NAME` to be invoked before the entry point of the executable is called (or immediately after the shared library is loaded if the procedure is linked in a shared library), in particular before the Ada run-time environment is set up.

Because of these specific contexts, the set of operations such a procedure can perform is very limited and the type of objects it can manipulate is essentially restricted to the elementary types. In particular, it must only contain code to which pragma Restrictions (No_Elaboration_Code) applies.

This pragma is used by GNAT to implement auto-initialization of shared Stand Alone Libraries, which provides a related capability without the restrictions listed above. Where possible, the use of Stand Alone Libraries is preferable to the use of this pragma.

2.102 Pragma Linker_Destructor

Syntax:

```
pragma Linker_Destructor (procedure_LOCAL_NAME);
```

`procedure_LOCAL_NAME` must refer to a parameterless procedure that is declared at the library level. A procedure to which this pragma is applied will be treated as a finalization routine by the linker. It is equivalent to `__attribute__((destructor))` in GNU C and causes `procedure_LOCAL_NAME` to be invoked after the entry point of the executable has exited (or immediately before the shared library is unloaded if the procedure is linked in a shared library), in particular after the Ada run-time environment is shut down.

See `pragma Linker_Constructor` for the set of restrictions that apply because of these specific contexts.

2.103 Pragma Linker_Section

Syntax:

```
pragma Linker_Section (
  [Entity  =>] LOCAL_NAME,
  [Section =>] static_string_EXPRESSION);
```

`LOCAL_NAME` must refer to an object, type, or subprogram that is declared at the library level. This pragma specifies the name of the linker section for the given entity. It is equivalent to `__attribute__((section))` in GNU C and causes `LOCAL_NAME` to be placed in the `static_string_EXPRESSION` section of the executable (assuming the linker doesn't rename the section). GNAT also provides an implementation defined aspect of the same name.

In the case of specifying this aspect for a type, the effect is to specify the corresponding section for all library-level objects of the type that do not have an explicit linker section set. Note that this only applies to whole objects, not to components of composite objects.

In the case of a subprogram, the linker section applies to all previously declared matching overloaded subprograms in the current declarative part which do not already have a linker section assigned. The linker section aspect is useful in this case for specifying different linker sections for different elements of such an overloaded set.

Note that an empty string specifies that no linker section is specified. This is not quite the same as omitting the pragma or aspect, since it can be used to specify that one element of an overloaded set of subprograms has the default linker section, or that one object of a type for which a linker section is specified should has the default linker section.

The compiler normally places library-level entities in standard sections depending on the class: procedures and functions generally go in the `.text` section, initialized variables in the `.data` section and uninitialized variables in the `.bss` section.

Other, special sections may exist on given target machines to map special hardware, for example I/O ports or flash memory. This pragma is a means to defer the final layout of the executable to the linker, thus fully working at the symbolic level with the compiler.

Some file formats do not support arbitrary sections so not all target machines support this pragma. The use of this pragma may cause a program execution to be erroneous if it is used to place an entity into an inappropriate section (e.g., a modified variable into the `.text` section). See also `pragma Persistent_BSS`.

```
--  Example of the use of pragma Linker_Section

package IO_Card is
  Port_A : Integer;
  pragma Volatile (Port_A);
  pragma Linker_Section (Port_A, ".bss.port_a");

  Port_B : Integer;
  pragma Volatile (Port_B);
  pragma Linker_Section (Port_B, ".bss.port_b");

  type Port_Type is new Integer with Linker_Section => ".bss";
  PA : Port_Type with Linker_Section => ".bss.PA";
  PB : Port_Type; --  ends up in linker section ".bss"

  procedure Q with Linker_Section => "Qsection";
end IO_Card;
```

2.104 Pragma Lock_Free

Syntax:

```
pragma Lock_Free [ (static_boolean_EXPRESSION) ];
```

This pragma may be specified for protected types or objects. It specifies that the implementation of protected operations must be implemented without locks. Compilation fails if the compiler cannot generate lock-free code for the operations.

The current conditions required to support this pragma are:

* Protected type declarations may not contain entries
* Protected subprogram declarations may not have nonelementary parameters

In addition, each protected subprogram body must satisfy:

* May reference only one protected component
* May not reference nonconstant entities outside the protected subprogram scope
* May not contain address representation items, allocators, or quantified expressions
* May not contain delay, goto, loop, or procedure-call statements
* May not contain exported and imported entities
* May not dereferenced access values
* Function calls and attribute references must be static

If the Lock_Free aspect is specified to be True for a protected unit and the Ceiling_Locking locking policy is in effect, then the run-time actions associated with the Ceiling_Locking locking policy (described in Ada RM D.3) are not performed when a protected operation of the protected unit is executed.

2.105 Pragma Loop_Invariant

Syntax:

```
pragma Loop_Invariant ( boolean_EXPRESSION );
```

The effect of this pragma is similar to that of pragma `Assert`, except that in an `Assertion_Policy` pragma, the identifier `Loop_Invariant` is used to control whether it is ignored or checked (or disabled).

`Loop_Invariant` can only appear as one of the items in the sequence of statements of a loop body, or nested inside block statements that appear in the sequence of statements of a loop body. The intention is that it be used to represent a "loop invariant" assertion, i.e. something that is true each time through the loop, and which can be used to show that the loop is achieving its purpose.

Multiple `Loop_Invariant` and `Loop_Variant` pragmas that apply to the same loop should be grouped in the same sequence of statements.

To aid in writing such invariants, the special attribute `Loop_Entry` may be used to refer to the value of an expression on entry to the loop. This attribute can only be used within the expression of a `Loop_Invariant` pragma. For full details, see documentation of attribute `Loop_Entry`.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.106 Pragma Loop_Optimize

Syntax:

```
pragma Loop_Optimize (OPTIMIZATION_HINT {, OPTIMIZATION_HINT});

OPTIMIZATION_HINT ::= Ivdep | No_Unroll | Unroll | No_Vector | Vector
```

This pragma must appear immediately within a loop statement. It allows the programmer to specify optimization hints for the enclosing loop. The hints are not mutually exclusive and can be freely mixed, but not all combinations will yield a sensible outcome.

There are five supported optimization hints for a loop:

* Ivdep

  The programmer asserts that there are no loop-carried dependencies which would prevent consecutive iterations of the loop from being executed simultaneously.

* No_Unroll

  The loop must not be unrolled. This is a strong hint: the compiler will not unroll a loop marked with this hint.

* Unroll

  The loop should be unrolled. This is a weak hint: the compiler will try to apply unrolling to this loop preferably to other optimizations, notably vectorization, but there is no guarantee that the loop will be unrolled.

* No_Vector

  The loop must not be vectorized. This is a strong hint: the compiler will not vectorize a loop marked with this hint.

* Vector

  The loop should be vectorized. This is a weak hint: the compiler will try to apply vectorization to this loop preferably to other optimizations, notably unrolling, but there is no guarantee that the loop will be vectorized.

These hints do not remove the need to pass the appropriate switches to the compiler in order to enable the relevant optimizations, that is to say '-funroll-loops' for unrolling and '-ftree-vectorize' for vectorization.

2.107 Pragma Loop_Variant

Syntax:

```
pragma Loop_Variant ( LOOP_VARIANT_ITEM {, LOOP_VARIANT_ITEM } );
LOOP_VARIANT_ITEM ::= CHANGE_DIRECTION => discrete_EXPRESSION
CHANGE_DIRECTION ::= Increases | Decreases
```

`Loop_Variant` can only appear as one of the items in the sequence of statements of a loop body, or nested inside block statements that appear in the sequence of statements of a loop body. It allows the specification of quantities which must always decrease or increase in successive iterations of the loop. In its simplest form, just one expression is specified, whose value must increase or decrease on each iteration of the loop.

In a more complex form, multiple arguments can be given which are interpreted in a nesting lexicographic manner. For example:

```
pragma Loop_Variant (Increases => X, Decreases => Y);
```

specifies that each time through the loop either X increases, or X stays the same and Y decreases. A `Loop_Variant` pragma ensures that the loop is making progress. It can be useful in helping to show informally or prove formally that the loop always terminates.

`Loop_Variant` is an assertion whose effect can be controlled using an `Assertion_Policy` with a check name of `Loop_Variant`. The policy can be `Check` to enable the loop variant check, `Ignore` to ignore the check (in which case the pragma has no effect on the program), or `Disable` in which case the pragma is not even checked for correct syntax.

Multiple `Loop_Invariant` and `Loop_Variant` pragmas that apply to the same loop should be grouped in the same sequence of statements.

The `Loop_Entry` attribute may be used within the expressions of the `Loop_Variant` pragma to refer to values on entry to the loop.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.108 Pragma Machine_Attribute

Syntax:

```
pragma Machine_Attribute (
     [Entity         =>] LOCAL_NAME,
     [Attribute_Name =>] static_string_EXPRESSION
  [, [Info           =>] static_EXPRESSION {, static_EXPRESSION}] );
```

Machine-dependent attributes can be specified for types and/or declarations. This pragma is semantically equivalent to `__attribute__((`*`attribute_name`*`))` (if `info` is not specified) or `__attribute__((`*`attribute_name(info)`*`))` or `__attribute__((`*`attribute_name(info,...)`*`))` in GNU C, where 'attribute_name' is recognized by the compiler middle-end or the `TARGET_ATTRIBUTE_TABLE` machine specific macro. Note that a string literal for the optional parameter `info` or the following ones is transformed by default into an identifier, which may make this pragma unusable for some attributes. For further information see *GNU Compiler Collection (GCC) Internals*.

2.109 Pragma Main

Syntax:

```
pragma Main
 (MAIN_OPTION [, MAIN_OPTION]);

MAIN_OPTION ::=
  [Stack_Size              =>] static_integer_EXPRESSION
| [Task_Stack_Size_Default =>] static_integer_EXPRESSION
| [Time_Slicing_Enabled    =>] static_boolean_EXPRESSION
```

This pragma is provided for compatibility with OpenVMS VAX Systems. It has no effect in GNAT, other than being syntax checked.

2.110 Pragma Main_Storage

Syntax:

```
pragma Main_Storage
```

```
    (MAIN_STORAGE_OPTION [, MAIN_STORAGE_OPTION]);

  MAIN_STORAGE_OPTION ::=
    [WORKING_STORAGE =>] static_SIMPLE_EXPRESSION
  | [TOP_GUARD       =>] static_SIMPLE_EXPRESSION
```

This pragma is provided for compatibility with OpenVMS VAX Systems. It has no effect in GNAT, other than being syntax checked.

2.111 Pragma Max_Queue_Length

Syntax:

```
pragma Max_Queue_Length (static_integer_EXPRESSION);
```

This pragma is used to specify the maximum callers per entry queue for individual protected entries and entry families. It accepts a single integer (-1 or more) as a parameter and must appear after the declaration of an entry.

A value of -1 represents no additional restriction on queue length.

2.112 Pragma No_Body

Syntax:

```
pragma No_Body;
```

There are a number of cases in which a package spec does not require a body, and in fact a body is not permitted. GNAT will not permit the spec to be compiled if there is a body around. The pragma No_Body allows you to provide a body file, even in a case where no body is allowed. The body file must contain only comments and a single No_Body pragma. This is recognized by the compiler as indicating that no body is logically present.

This is particularly useful during maintenance when a package is modified in such a way that a body needed before is no longer needed. The provision of a dummy body with a No_Body pragma ensures that there is no interference from earlier versions of the package body.

2.113 Pragma No_Caching

Syntax:

```
pragma No_Caching [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `No_Caching` in the SPARK 2014 Reference Manual, section 7.1.2.

2.114 Pragma No_Component_Reordering

Syntax:

```
pragma No_Component_Reordering [([Entity =>] type_LOCAL_NAME)];
```

`type_LOCAL_NAME` must refer to a record type declaration in the current declarative part. The effect is to preclude any reordering of components for the layout of the record, i.e. the record is laid out by the compiler in the order in which the components are declared textually. The form with no argument is a configuration pragma which applies to all record types declared in units to which the pragma applies and there is a requirement that this pragma be used consistently within a partition.

2.115 Pragma No_Elaboration_Code_All

Syntax:

```
pragma No_Elaboration_Code_All [(program_unit_NAME)];
```

This is a program unit pragma (there is also an equivalent aspect of the same name) that establishes the restriction `No_Elaboration_Code` for the current unit and any extended main source units (body and subunits). It also has the effect of enforcing a transitive application of this aspect, so that if any unit is implicitly or explicitly with'ed by the current unit, it must also have the *No_Elaboration_Code_All* aspect set. It may be applied to package or subprogram specs or their generic versions.

2.116 Pragma No_Heap_Finalization

Syntax:

```
pragma No_Heap_Finalization [ (first_subtype_LOCAL_NAME) ];
```

Pragma `No_Heap_Finalization` may be used as a configuration pragma or as a type-specific pragma.

In its configuration form, the pragma must appear within a configuration file such as gnat.adc, without an argument. The pragma suppresses the call to `Finalize` for heap-allocated objects created through library-level named access-to-object types in cases where the designated type requires finalization actions.

In its type-specific form, the argument of the pragma must denote a library-level named access-to-object type. The pragma suppresses the call to `Finalize` for heap-allocated objects created through the specific access type in cases where the designated type requires finalization actions.

It is still possible to finalize such heap-allocated objects by explicitly deallocating them.

A library-level named access-to-object type declared within a generic unit will lose its `No_Heap_Finalization` pragma when the corresponding instance does not appear at the library level.

2.117 Pragma No_Inline

Syntax:

```
pragma No_Inline (NAME {, NAME});
```

This pragma suppresses inlining for the callable entity or the instances of the generic subprogram designated by `NAME`, including inlining that results from the use of pragma `Inline`. This pragma is always active, in particular it is not subject to the use of option '-gnatn' or '-gnatN'. It is illegal to specify both pragma `No_Inline` and pragma `Inline_Always` for the same `NAME`.

2.118 Pragma No_Raise

Syntax:

```
pragma No_Raise (subprogram_LOCAL_NAME {, subprogram_LOCAL_NAME});
```

Each `subprogram_LOCAL_NAME` argument must refer to one or more subprogram declarations in the current declarative part. A subprogram to which this pragma is applied may not raise an exception that is not caught within it. An implementation-defined check named *Raise_Check* is associated with the pragma, and *Program_Error* is raised upon its failure (see RM 11.5(19/5)).

2.119 Pragma No_Return

Syntax:

```
pragma No_Return (procedure_LOCAL_NAME {, procedure_LOCAL_NAME});
```

Each `procedure_LOCAL_NAME` argument must refer to one or more procedure declarations in the current declarative part. A procedure to which this pragma is applied may not contain any explicit `return` statements. In addition, if the procedure contains any implicit returns from falling off the end of a statement sequence, then execution of that implicit return will cause Program_Error to be raised.

One use of this pragma is to identify procedures whose only purpose is to raise an exception. Another use of this pragma is to suppress incorrect warnings about missing returns in functions, where the last statement of a function statement sequence is a call to such a procedure.

Note that in Ada 2005 mode, this pragma is part of the language. It is available in all earlier versions of Ada as an implementation-defined pragma.

2.120 Pragma No_Strict_Aliasing

Syntax:

```
pragma No_Strict_Aliasing [([Entity =>] type_LOCAL_NAME)];
```

`type_LOCAL_NAME` must refer to an access type declaration in the current declarative part. The effect is to inhibit strict aliasing optimization for the given type. The form with no arguments is a configuration pragma which applies to all access types declared in units to which the pragma applies. For a detailed description of the strict aliasing optimization, and the situations in which it must be suppressed, see the section on Optimization and Strict Aliasing in the *GNAT User's Guide*.

This pragma currently has no effects on access to unconstrained array types.

2.121 Pragma No_Tagged_Streams

Syntax:

```
pragma No_Tagged_Streams [([Entity =>] tagged_type_LOCAL_NAME)];
```

Normally when a tagged type is introduced using a full type declaration, part of the processing includes generating stream access routines to be used by stream attributes referencing the type (or one of its subtypes or derived types). This can involve the generation of significant amounts of code which is wasted space if stream routines are not needed for the type in question.

The `No_Tagged_Streams` pragma causes the generation of these stream routines to be skipped, and any attempt to use stream operations on types subject to this pragma will be statically rejected as illegal.

There are two forms of the pragma. The form with no arguments must appear in a declarative sequence or in the declarations of a package spec. This pragma affects all subsequent root tagged types declared in the declaration sequence, and specifies that no stream routines be generated. The form with an argument (for which there is also a corresponding aspect) specifies a single root tagged type for which stream routines are not to be generated.

Once the pragma has been given for a particular root tagged type, all subtypes and derived types of this type inherit the pragma automatically, so the effect applies to a complete hierarchy (this is necessary to deal with the class-wide dispatching versions of the stream routines).

When pragmas `Discard_Names` and `No_Tagged_Streams` are simultaneously applied to a tagged type its Expanded_Name and External_Tag are initialized with empty strings. This is useful to avoid exposing entity names at binary level but has a negative impact on the debuggability of tagged types.

2.122 Pragma Normalize_Scalars

Syntax:

```
pragma Normalize_Scalars;
```

This is a language defined pragma which is fully implemented in GNAT. The effect is to cause all scalar objects that are not otherwise initialized to be initialized. The initial values are implementation dependent and are as follows:

'Standard.Character'
: Objects whose root type is Standard.Character are initialized to Character'Last unless the subtype range excludes NUL (in which case NUL is used). This choice will always generate an invalid value if one exists.

'Standard.Wide_Character'
: Objects whose root type is Standard.Wide_Character are initialized to Wide_Character'Last unless the subtype range excludes NUL (in which case NUL is used). This choice will always generate an invalid value if one exists.

'Standard.Wide_Wide_Character'
: Objects whose root type is Standard.Wide_Wide_Character are initialized to the invalid value 16#FFFF_FFFF# unless the subtype range excludes NUL (in which case NUL is used). This choice will always generate an invalid value if one exists.

'Integer types'
: Objects of an integer type are treated differently depending on whether negative values are present in the subtype. If no negative values are present, then all one bits is used as the initial value except in the special case where zero is excluded from the subtype, in which case all zero bits are used. This choice will always generate an invalid value if one exists.

: For subtypes with negative values present, the largest negative number is used, except in the unusual case where this largest negative number is in the subtype, and the largest positive number is not, in which case the largest positive value is used. This choice will always generate an invalid value if one exists.

'Floating-Point Types'
: Objects of all floating-point types are initialized to all 1-bits. For standard IEEE format, this corresponds to a NaN (not a number) which is indeed an invalid value.

'Fixed-Point Types'
: Objects of all fixed-point types are treated as described above for integers, with the rules applying to the underlying integer value used to represent the fixed-point value.

'Modular types'
: Objects of a modular type are initialized to all one bits, except in the special case where zero is excluded from the subtype, in which case all zero bits are used. This choice will always generate an invalid value if one exists.

'Enumeration types'
: Objects of an enumeration type are initialized to all one-bits, i.e., to the value `2 ** typ'Size - 1` unless the subtype excludes the literal whose Pos value is zero, in which case a code of zero is used. This choice will always generate an invalid value if one exists.

2.123 Pragma Obsolescent

Syntax:

```
pragma Obsolescent;

pragma Obsolescent (
  [Message =>] static_string_EXPRESSION
[,[Version =>] Ada_05]);

pragma Obsolescent (
  [Entity  =>] NAME
[,[Message =>] static_string_EXPRESSION
[,[Version =>] Ada_05]]);
```

This pragma can occur immediately following a declaration of an entity, including the case of a record component. If no Entity argument is present, then this declaration is the one to which the pragma applies. If an Entity parameter is present, it must either match the name of the entity in this declaration, or alternatively, the pragma can immediately follow an enumeration type declaration, where the Entity argument names one of the enumeration literals.

This pragma is used to indicate that the named entity is considered obsolescent and should not be used. Typically this is used when an API must be modified by eventually removing or modifying existing subprograms or other entities. The pragma can be used at an intermediate stage when the entity is still present, but will be removed later.

The effect of this pragma is to output a warning message on a reference to an entity thus marked that the subprogram is obsolescent if the appropriate warning option in the compiler is activated. If the `Message` parameter is present, then a second warning message is given containing this text. In addition, a reference to the entity is considered to be a violation of pragma `Restrictions (No_Obsolescent_Features)`.

This pragma can also be used as a program unit pragma for a package, in which case the entity name is the name of the package, and the pragma indicates that the entire package is considered obsolescent. In this case a client `with`ing such a package violates the restriction, and the `with` clause is flagged with warnings if the warning option is set.

If the `Version` parameter is present (which must be exactly the identifier `Ada_05`, no other argument is allowed), then the indication of obsolescence applies only when compiling in Ada 2005 mode. This is primarily intended for dealing with the situations in the predefined library where subprograms or packages have become defined as obsolescent in Ada 2005 (e.g., in `Ada.Characters.Handling`), but may be used anywhere.

The following examples show typical uses of this pragma:

```
package p is
   pragma Obsolescent (p, Message => "use pp instead of p");
end p;

package q is
   procedure q2;
   pragma Obsolescent ("use q2new instead");

   type R is new integer;
   pragma Obsolescent
     (Entity  => R,
      Message => "use RR in Ada 2005",
      Version => Ada_05);

   type M is record
      F1 : Integer;
      F2 : Integer;
      pragma Obsolescent;
      F3 : Integer;
   end record;

   type E is (a, bc, 'd', quack);
   pragma Obsolescent (Entity => bc)
   pragma Obsolescent (Entity => 'd')

   function "+"
     (a, b : character) return character;
   pragma Obsolescent (Entity => "+");
end;
```

Note that, as for all pragmas, if you use a pragma argument identifier, then all subsequent parameters must also use a pragma argument identifier. So if you specify `Entity =>` for the `Entity` argument, and a `Message` argument is present, it must be preceded by `Message =>`.

2.124 Pragma Optimize_Alignment

Syntax:

```
pragma Optimize_Alignment (TIME | SPACE | OFF);
```

This is a configuration pragma which affects the choice of default alignments for types and objects where no alignment is explicitly specified. There is a time/space trade-off in the selection of these values. Large alignments result in more efficient code, at the expense of larger data space, since sizes have to be increased to match these alignments. Smaller alignments save space, but the access code is slower. The normal choice of default alignments for types and individual alignment promotions for objects (which is what you get if you do not use this pragma, or if you use an argument of OFF), tries to balance these two requirements.

Specifying SPACE causes smaller default alignments to be chosen in two cases. First any packed record is given an alignment of 1. Second, if a size is given for the type, then the alignment is chosen to avoid increasing this size. For example, consider:

```
type R is record
   X : Integer;
   Y : Character;
end record;

for R'Size use 5*8;
```

In the default mode, this type gets an alignment of 4, so that access to the Integer field X are efficient. But this means that objects of the type end up with a size of 8 bytes. This is a valid choice, since sizes of objects are allowed to be bigger than the size of the type, but it can waste space if for example fields of type R appear in an enclosing record. If the above type is compiled in `Optimize_Alignment (Space)` mode, the alignment is set to 1.

However, there is one case in which SPACE is ignored. If a variable length record (that is a discriminated record with a component which is an array whose length depends on a discriminant), has a pragma Pack, then it is not in general possible to set the alignment of such a record to one, so the pragma is ignored in this case (with a warning).

Specifying SPACE also disables alignment promotions for standalone objects, which occur when the compiler increases the alignment of a specific object without changing the alignment of its type.

Specifying SPACE also disables component reordering in unpacked record types, which can result in larger sizes in order to meet alignment requirements.

Specifying TIME causes larger default alignments to be chosen in the case of small types with sizes that are not a power of 2. For example, consider:

```
type R is record
   A : Character;
   B : Character;
   C : Boolean;
end record;

pragma Pack (R);
for R'Size use 17;
```

The default alignment for this record is normally 1, but if this type is compiled in `Optimize_Alignment (Time)` mode, then the alignment is set to 4, which wastes space for objects of the type, since they are now 4 bytes long, but results in more efficient access when the whole record is referenced.

As noted above, this is a configuration pragma, and there is a requirement that all units in a partition be compiled with a consistent setting of the optimization setting. This would normally be achieved by use of a configuration pragma file containing the appropriate setting. The exception to this rule is that units with an explicit configuration pragma in the same file as the source unit are excluded from the consistency check, as are all predefined units. The latter are compiled by default in pragma Optimize_Alignment (Off) mode if no pragma appears at the start of the file.

2.125 Pragma Ordered

Syntax:

```
pragma Ordered (enumeration_first_subtype_LOCAL_NAME);
```

Most enumeration types are from a conceptual point of view unordered. For example, consider:

```
type Color is (Red, Blue, Green, Yellow);
```

By Ada semantics `Blue > Red` and `Green > Blue`, but really these relations make no sense; the enumeration type merely specifies a set of possible colors, and the order is unimportant.

For unordered enumeration types, it is generally a good idea if clients avoid comparisons (other than equality or inequality) and explicit ranges. (A 'client' is a unit where the type is referenced, other than the unit where the type is declared, its body, and its subunits.) For example, if code buried in some client says:

```
if Current_Color < Yellow then ...
if Current_Color in Blue .. Green then ...
```

then the client code is relying on the order, which is undesirable. It makes the code hard to read and creates maintenance difficulties if entries have to be added to the enumeration type. Instead, the code in the client should list the possibilities, or an appropriate subtype should be declared in the unit that declares the original enumeration type. E.g., the following subtype could be declared along with the type `Color`:

```
subtype RBG is Color range Red .. Green;
```

and then the client could write:

```
if Current_Color in RBG then ...
if Current_Color = Blue or Current_Color = Green then ...
```

However, some enumeration types are legitimately ordered from a conceptual point of view. For example, if you declare:

```
type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

then the ordering imposed by the language is reasonable, and clients can depend on it, writing for example:

```
if D in Mon .. Fri then ...
if D < Wed then ...
```

The pragma 'Ordered' is provided to mark enumeration types that are conceptually ordered, alerting the reader that clients may depend on the ordering. GNAT provides a pragma to mark enumerations as ordered rather than one to mark them as unordered, since in our experience, the great majority of enumeration types are conceptually unordered.

The types `Boolean`, `Character`, `Wide_Character`, and `Wide_Wide_Character` are considered to be ordered types, so each is declared with a pragma `Ordered` in package `Standard`.

Normally pragma `Ordered` serves only as documentation and a guide for coding standards, but GNAT provides a warning switch '-gnatw.u' that requests warnings for inappropriate uses (comparisons and explicit subranges) for unordered types. If this switch is used, then any enumeration type not marked with pragma `Ordered` will be considered as unordered, and will generate warnings for inappropriate uses.

Note that generic types are not considered ordered or unordered (since the template can be instantiated for both cases), so we never generate warnings for the case of generic enumerated types.

For additional information please refer to the description of the '-gnatw.u' switch in the GNAT User's Guide.

2.126 Pragma Overflow_Mode

Syntax:

```
pragma Overflow_Mode
 (  [General    =>] MODE
  [,[Assertions =>] MODE]);

MODE ::= STRICT | MINIMIZED | ELIMINATED
```

This pragma sets the current overflow mode to the given setting. For details of the meaning of these modes, please refer to the 'Overflow Check Handling in GNAT' appendix in the GNAT User's Guide. If only the `General` parameter is present, the given mode applies to all expressions. If both parameters are present, the `General` mode applies to expressions outside assertions, and the `Eliminated` mode applies to expressions within assertions.

The case of the `MODE` parameter is ignored, so `MINIMIZED`, `Minimized` and `minimized` all have the same effect.

The `Overflow_Mode` pragma has the same scoping and placement rules as pragma `Suppress`, so it can occur either as a configuration pragma, specifying a default for the whole program, or in a declarative scope, where it applies to the remaining declarations and statements in that scope.

The pragma `Suppress (Overflow_Check)` suppresses overflow checking, but does not affect the overflow mode.

The pragma `Unsuppress (Overflow_Check)` unsuppresses (enables) overflow checking, but does not affect the overflow mode.

2.127 Pragma Overriding_Renamings

Syntax:

```
pragma Overriding_Renamings;
```

This is a GNAT configuration pragma to simplify porting legacy code accepted by the Rational Ada compiler. In the presence of this pragma, a renaming declaration that renames an inherited operation declared in the same scope is legal if selected notation is used as in:

```
pragma Overriding_Renamings;
...
package R is
  function F (..);
  ...
  function F (..) renames R.F;
end R;
```

even though RM 8.3 (15) stipulates that an overridden operation is not visible within the declaration of the overriding operation.

2.128 Pragma Part_Of

Syntax:

```
pragma Part_Of (ABSTRACT_STATE);

ABSTRACT_STATE ::= NAME
```

For the semantics of this pragma, see the entry for aspect `Part_Of` in the SPARK 2014 Reference Manual, section 7.2.6.

2.129 Pragma Partition_Elaboration_Policy

Syntax:

```
pragma Partition_Elaboration_Policy (POLICY_IDENTIFIER);

POLICY_IDENTIFIER ::= Concurrent | Sequential
```

This pragma is standard in Ada 2005, but is available in all earlier versions of Ada as an implementation-defined pragma. See Ada 2012 Reference Manual for details.

2.130 Pragma Passive

Syntax:

```
pragma Passive [(Semaphore | No)];
```

Syntax checked, but otherwise ignored by GNAT. This is recognized for compatibility with DEC Ada 83 implementations, where it is used within a task definition to request that a task be made passive. If the argument `Semaphore` is present, or the argument is omitted, then DEC Ada 83 treats the pragma as an assertion that the containing task is passive and that optimization of context switch with this task is permitted and desired. If the argument `No` is present, the task must not be optimized. GNAT does not attempt to optimize any tasks in this manner (since protected objects are available in place of passive tasks).

For more information on the subject of passive tasks, see the section 'Passive Task Optimization' in the GNAT Users Guide.

2.131 Pragma Persistent_BSS

Syntax:

```
pragma Persistent_BSS [(object_LOCAL_NAME)]
```

This pragma allows selected objects to be placed in the `.persistent_bss` section. On some targets the linker and loader provide for special treatment of this section, allowing a program to be reloaded without affecting the contents of this data (hence the name persistent).

There are two forms of usage. If an argument is given, it must be the local name of a library-level object, with no explicit initialization and whose type is potentially persistent. If no argument is given, then the pragma is a configuration pragma, and applies to all library-level objects with no explicit initialization of potentially persistent types.

A potentially persistent type is a scalar type, or an untagged, non-discriminated record, all of whose components have no explicit initialization and are themselves of a potentially persistent type, or an array, all of whose constraints are static, and whose component type is potentially persistent.

If this pragma is used on a target where this feature is not supported, then the pragma will be ignored. See also `pragma Linker_Section`.

2.132 Pragma Post

Syntax:

```
pragma Post (Boolean_Expression);
```

The `Post` pragma is intended to be an exact replacement for the language-defined `Post` aspect, and shares its restrictions and semantics. It must appear either immediately following the corresponding subprogram declaration (only other pragmas may intervene), or if there is no separate subprogram declaration, then it can appear at the start of the declarations in a subprogram body (preceded only by other pragmas).

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.133 Pragma Postcondition

Syntax:

```
pragma Postcondition (
   [Check   =>] Boolean_Expression
 [,[Message =>] String_Expression]);
```

The `Postcondition` pragma allows specification of automatic postcondition checks for subprograms. These checks are similar to assertions, but are automatically inserted just prior to the return statements of the subprogram with which they are associated (including implicit returns at the end of procedure bodies and associated exception handlers).

In addition, the boolean expression which is the condition which must be true may contain references to function'Result in the case of a function to refer to the returned value.

`Postcondition` pragmas may appear either immediately following the (separate) declaration of a subprogram, or at the start of the declarations of a subprogram body. Only other pragmas may intervene (that is appear between the subprogram declaration and its postconditions, or appear before the postcondition in the declaration sequence in a subprogram body). In the case of a postcondition appearing after a subprogram declaration, the formal arguments of the subprogram are visible, and can be referenced in the postcondition expressions.

The postconditions are collected and automatically tested just before any return (implicit or explicit) in the subprogram body. A postcondition is only recognized if postconditions are active at the time the pragma is encountered. The compiler switch 'gnata' turns on all postconditions by default, and pragma `Check_Policy` with an identifier of `Postcondition` can also be used to control whether postconditions are active.

The general approach is that postconditions are placed in the spec if they represent functional aspects which make sense to the client. For example we might have:

```
function Direction return Integer;
pragma Postcondition
 (Direction'Result = +1
    or else
   Direction'Result = -1);
```

which serves to document that the result must be `+1` or `-1`, and will test that this is the case at run time if postcondition checking is active.

Postconditions within the subprogram body can be used to check that some internal aspect of the implementation, not visible to the client, is operating as expected. For instance if a square root routine keeps an internal counter of the number of times it is called, then we might have the following postcondition:

```
Sqrt_Calls : Natural := 0;

function Sqrt (Arg : Float) return Float is
  pragma Postcondition
    (Sqrt_Calls = Sqrt_Calls'Old + 1);
  ...
end Sqrt
```

As this example, shows, the use of the `Old` attribute is often useful in postconditions to refer to the state on entry to the subprogram.

Note that postconditions are only checked on normal returns from the subprogram. If an abnormal return results from raising an exception, then the postconditions are not checked.

If a postcondition fails, then the exception `System.Assertions.Assert_Failure` is raised. If a message argument was supplied, then the given string will be used as the exception message. If no message argument was supplied, then the default message has the form "Postcondition failed at file_name:line". The exception is raised in the context of the subprogram body, so it is possible to catch postcondition failures within the subprogram body itself.

Within a package spec, normal visibility rules in Ada would prevent forward references within a postcondition pragma to functions defined later in the same package. This would introduce undesirable ordering constraints. To avoid this problem, all postcondition pragmas are analyzed at the end of the package spec, allowing forward references.

The following example shows that this even allows mutually recursive postconditions as in:

```
package Parity_Functions is
```

```
    function Odd  (X : Natural) return Boolean;
    pragma Postcondition
      (Odd'Result =
         (x = 1
           or else
         (x /= 0 and then Even (X - 1))));

    function Even (X : Natural) return Boolean;
    pragma Postcondition
      (Even'Result =
         (x = 0
           or else
         (x /= 1 and then Odd (X - 1))));

end Parity_Functions;
```

There are no restrictions on the complexity or form of conditions used within `Postcondition` pragmas. The following example shows that it is even possible to verify performance behavior.

```
package Sort is

   Performance : constant Float;
   --  Performance constant set by implementation
   --  to match target architecture behavior.

   procedure Treesort (Arg : String);
   --  Sorts characters of argument using N*logN sort
   pragma Postcondition
     (Float (Clock - Clock'Old) <=
        Float (Arg'Length) *
        log (Float (Arg'Length)) *
        Performance);
end Sort;
```

Note: postcondition pragmas associated with subprograms that are marked as Inline_Always, or those marked as Inline with front-end inlining (-gnatN option set) are accepted and legality-checked by the compiler, but are ignored at run-time even if postcondition checking is enabled.

Note that pragma `Postcondition` differs from the language-defined `Post` aspect (and corresponding `Post` pragma) in allowing multiple occurrences, allowing occurences in the body even if there is a separate spec, and allowing a second string parameter, and the use of the pragma identifier `Check`. Historically, pragma `Postcondition` was implemented prior to the development of Ada 2012, and has been retained in its original form for compatibility purposes.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.134 Pragma Post_Class

Syntax:

```
pragma Post_Class (Boolean_Expression);
```

The `Post_Class` pragma is intended to be an exact replacement for the language-defined `Post'Class` aspect, and shares its restrictions and semantics. It must appear either immediately following the corresponding subprogram declaration (only other pragmas may intervene), or if there is no separate subprogram declaration, then it can appear at the start of the declarations in a subprogram body (preceded only by other pragmas).

Note: This pragma is called `Post_Class` rather than `Post'Class` because the latter would not be strictly conforming to the allowed syntax for pragmas. The motivation for providing pragmas equivalent to the aspects is to allow a program to be written using the pragmas, and then compiled if necessary using an Ada compiler that does not recognize the pragmas or aspects, but is prepared to ignore the pragmas. The assertion policy that controls this pragma is `Post'Class`, not `Post_Class`.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.135 Pragma Pre

Syntax:

```
pragma Pre (Boolean_Expression);
```

The `Pre` pragma is intended to be an exact replacement for the language-defined `Pre` aspect, and shares its restrictions and semantics. It must appear either immediately following the corresponding subprogram declaration (only other pragmas may intervene), or if there is no separate subprogram declaration, then it can appear at the start of the declarations in a subprogram body (preceded only by other pragmas).

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.136 Pragma Precondition

Syntax:

```
pragma Precondition (
   [Check   =>] Boolean_Expression
 [,[Message =>] String_Expression]);
```

The `Precondition` pragma is similar to `Postcondition` except that the corresponding checks take place immediately upon entry to the subprogram, and if a precondition fails, the exception is raised in the context of the caller, and the attribute 'Result cannot be used within the precondition expression.

Otherwise, the placement and visibility rules are identical to those described for postconditions. The following is an example of use within a package spec:

```
package Math_Functions is
   ...
   function Sqrt (Arg : Float) return Float;
   pragma Precondition (Arg >= 0.0)
```

```
   ...
end Math_Functions;
```

`Precondition` pragmas may appear either immediately following the (separate) declaration of a subprogram, or at the start of the declarations of a subprogram body. Only other pragmas may intervene (that is appear between the subprogram declaration and its postconditions, or appear before the postcondition in the declaration sequence in a subprogram body).

Note: precondition pragmas associated with subprograms that are marked as Inline_Always, or those marked as Inline with front-end inlining (-gnatN option set) are accepted and legality-checked by the compiler, but are ignored at run-time even if precondition checking is enabled.

Note that pragma `Precondition` differs from the language-defined `Pre` aspect (and corresponding `Pre` pragma) in allowing multiple occurrences, allowing occurences in the body even if there is a separate spec, and allowing a second string parameter, and the use of the pragma identifier `Check`. Historically, pragma `Precondition` was implemented prior to the development of Ada 2012, and has been retained in its original form for compatibility purposes.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.137 Pragma Predicate

Syntax:

```
pragma Predicate
  ([Entity =>] type_LOCAL_NAME,
   [Check  =>] EXPRESSION);
```

This pragma (available in all versions of Ada in GNAT) encompasses both the `Static_Predicate` and `Dynamic_Predicate` aspects in Ada 2012. A predicate is regarded as static if it has an allowed form for `Static_Predicate` and is otherwise treated as a `Dynamic_Predicate`. Otherwise, predicates specified by this pragma behave exactly as described in the Ada 2012 reference manual. For example, if we have

```
type R is range 1 .. 10;
subtype S is R;
pragma Predicate (Entity => S, Check => S not in 4 .. 6);
subtype Q is R
pragma Predicate (Entity => Q, Check => F(Q) or G(Q));
```

the effect is identical to the following Ada 2012 code:

```
type R is range 1 .. 10;
subtype S is R with
  Static_Predicate => S not in 4 .. 6;
subtype Q is R with
  Dynamic_Predicate => F(Q) or G(Q);
```

Note that there are no pragmas `Dynamic_Predicate` or `Static_Predicate`. That is because these pragmas would affect legality and semantics of the program and thus do not have a neutral effect if ignored. The motivation behind providing pragmas equivalent to corresponding aspects is to allow a program to be written using the pragmas, and then compiled with a compiler that will ignore the pragmas. That doesn't work in the case of static and dynamic predicates, since if the corresponding pragmas are ignored, then the behavior of the program is fundamentally changed (for example a membership test `A in B` would not take into account a predicate defined for subtype B). When following this approach, the use of predicates should be avoided.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.138 Pragma Predicate_Failure

Syntax:

```
pragma Predicate_Failure
  ([Entity  =>] type_LOCAL_NAME,
   [Message =>] String_Expression);
```

The `Predicate_Failure` pragma is intended to be an exact replacement for the language-defined `Predicate_Failure` aspect, and shares its restrictions and semantics.

2.139 Pragma Preelaborable_Initialization

Syntax:

```
pragma Preelaborable_Initialization (DIRECT_NAME);
```

This pragma is standard in Ada 2005, but is available in all earlier versions of Ada as an implementation-defined pragma. See Ada 2012 Reference Manual for details.

2.140 Pragma Prefix_Exception_Messages

Syntax:

```
pragma Prefix_Exception_Messages;
```

This is an implementation-defined configuration pragma that affects the behavior of raise statements with a message given as a static string constant (typically a string literal). In such cases, the string will be automatically prefixed by the name of the enclosing entity (giving the package and subprogram containing the raise statement). This helps to identify where messages are coming from, and this mode is automatic for the run-time library.

The pragma has no effect if the message is computed with an expression other than a static string constant, since the assumption in this case is that the program computes exactly the string it wants. If you still want the prefixing in this case, you can always call `GNAT.Source_Info.Enclosing_Entity` and prepend the string manually.

2.141 Pragma Pre_Class

Syntax:

```
pragma Pre_Class (Boolean_Expression);
```

The `Pre_Class` pragma is intended to be an exact replacement for the language-defined `Pre'Class` aspect, and shares its restrictions and semantics. It must appear either immediately following the corresponding subprogram declaration (only other pragmas may intervene), or if there is no separate subprogram declaration, then it can appear at the start of the declarations in a subprogram body (preceded only by other pragmas).

Note: This pragma is called `Pre_Class` rather than `Pre'Class` because the latter would not be strictly conforming to the allowed syntax for pragmas. The motivation for providing pragmas equivalent to the aspects is to allow a program to be written using the pragmas, and then compiled if necessary using an Ada compiler that does not recognize the pragmas or aspects, but is prepared to ignore the pragmas. The assertion policy that controls this pragma is `Pre'Class`, not `Pre_Class`.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.142 Pragma Priority_Specific_Dispatching

Syntax:

```
pragma Priority_Specific_Dispatching (
   POLICY_IDENTIFIER,
   first_priority_EXPRESSION,
   last_priority_EXPRESSION)

POLICY_IDENTIFIER ::=
   EDF_Across_Priorities            |
   FIFO_Within_Priorities           |
   Non_Preemptive_Within_Priorities |
   Round_Robin_Within_Priorities
```

This pragma is standard in Ada 2005, but is available in all earlier versions of Ada as an implementation-defined pragma. See Ada 2012 Reference Manual for details.

2.143 Pragma Profile

Syntax:

```
pragma Profile (Ravenscar | Restricted | Rational | Jorvik |
                GNAT_Extended_Ravenscar | GNAT_Ravenscar_EDF );
```

This pragma is standard in Ada 2005, but is available in all earlier versions of Ada as an implementation-defined pragma. This is a configuration pragma that establishes a set of configuration pragmas that depend on the argument. `Ravenscar` is standard in Ada 2005. `Jorvik` is standard in Ada 202x. The other possibilities (`Restricted`, `Rational`, `GNAT_Extended_Ravenscar`, `GNAT_Ravenscar_EDF`) are implementation-defined. `GNAT_Extended_Ravenscar` is an alias for `Jorvik`.

The set of configuration pragmas is defined in the following sections.

* Pragma Profile (Ravenscar)

  The `Ravenscar` profile is standard in Ada 2005, but is available in all earlier versions of Ada as an implementation-defined pragma. This profile establishes the following set of configuration pragmas:

  * `Task_Dispatching_Policy (FIFO_Within_Priorities)`

    [RM D.2.2] Tasks are dispatched following a preemptive priority-ordered scheduling policy.

* `Locking_Policy (Ceiling_Locking)`

  [RM D.3] While tasks and interrupts execute a protected action, they inherit the ceiling priority of the corresponding protected object.
* `Detect_Blocking`

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

plus the following set of restrictions:

* `Max_Entry_Queue_Length => 1`

  No task can be queued on a protected entry.
* `Max_Protected_Entries => 1`
* `Max_Task_Entries => 0`

  No rendezvous statements are allowed.
* `No_Abort_Statements`
* `No_Dynamic_Attachment`
* `No_Dynamic_Priorities`
* `No_Implicit_Heap_Allocations`
* `No_Local_Protected_Objects`
* `No_Local_Timing_Events`
* `No_Protected_Type_Allocators`
* `No_Relative_Delay`
* `No_Requeue_Statements`
* `No_Select_Statements`
* `No_Specific_Termination_Handlers`
* `No_Task_Allocators`
* `No_Task_Hierarchy`
* `No_Task_Termination`
* `Simple_Barriers`

The Ravenscar profile also includes the following restrictions that specify that there are no semantic dependencies on the corresponding predefined packages:

* `No_Dependence => Ada.Asynchronous_Task_Control`
* `No_Dependence => Ada.Calendar`
* `No_Dependence => Ada.Execution_Time.Group_Budget`
* `No_Dependence => Ada.Execution_Time.Timers`
* `No_Dependence => Ada.Task_Attributes`
* `No_Dependence => System.Multiprocessors.Dispatching_Domains`

This set of configuration pragmas and restrictions correspond to the definition of the 'Ravenscar Profile' for limited tasking, devised and published by the *International Real-Time Ada Workshop, 1997*. A description is also available at '`http://www-users.cs.york.ac.uk/~burns/ravenscar.ps`'.

The original definition of the profile was revised at subsequent IRTAW meetings. It has been included in the ISO *Guide for the Use of the Ada Programming Language in High Integrity Systems*, and was made part of the Ada 2005 standard. The formal definition given by the Ada Rapporteur Group (ARG) can be found in two Ada Issues (AI-249 and AI-305) available at '`http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00249.txt`' and '`http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00305.txt`'.

The above set is a superset of the restrictions provided by pragma `Profile (Restricted)`, it includes six additional restrictions (`Simple_Barriers`, `No_Select_Statements`, `No_Calendar`, `No_Implicit_Heap_Allocations`, `No_Relative_Delay` and `No_Task_Termination`). This means that pragma `Profile (Ravenscar)`, like the pragma `Profile (Restricted)`, automatically causes the use of a simplified, more efficient version of the tasking run-time library.

* Pragma Profile (Jorvik)

  `Jorvik` is the new profile added to the Ada 202x draft standard, previously implemented under the name `GNAT_Extended_Ravenscar`.

  The `No_Implicit_Heap_Allocations` restriction has been replaced by `No_Implicit_Task_Allocations` and `No_Implicit_Protected_Object_Allocations`.

  The `Simple_Barriers` restriction has been replaced by `Pure_Barriers`.

  The `Max_Protected_Entries`, `Max_Entry_Queue_Length`, and `No_Relative_Delay` restrictions have been removed.

  Details on the rationale for `Jorvik` and implications for use may be found in *A New Ravenscar-Based Profile* by P. Rogers, J. Ruiz, T. Gingold and P. Bernardi, in *Reliable Software Technologies – Ada Europe 2017*, Springer-Verlag Lecture Notes in Computer Science, Number 10300.

* Pragma Profile (GNAT_Ravenscar_EDF)

  This profile corresponds to the Ravenscar profile but using EDF_Across_Priority as the Task_Scheduling_Policy.

* Pragma Profile (Restricted)

  This profile corresponds to the GNAT restricted run time. It establishes the following set of restrictions:

  * `No_Abort_Statements`
  * `No_Entry_Queue`
  * `No_Task_Hierarchy`
  * `No_Task_Allocators`
  * `No_Dynamic_Priorities`
  * `No_Terminate_Alternatives`
  * `No_Dynamic_Attachment`
  * `No_Protected_Type_Allocators`
  * `No_Local_Protected_Objects`
  * `No_Requeue_Statements`
  * `No_Task_Attributes_Package`

  * `Max_Asynchronous_Select_Nesting = 0`
  * `Max_Task_Entries = 0`
  * `Max_Protected_Entries = 1`
  * `Max_Select_Alternatives = 0`

  This set of restrictions causes the automatic selection of a simplified version of the run time that provides improved performance for the limited set of tasking functionality permitted by this set of restrictions.

* Pragma Profile (Rational)

  The Rational profile is intended to facilitate porting legacy code that compiles with the Rational APEX compiler, even when the code includes non- conforming Ada constructs. The profile enables the following three pragmas:

  * `pragma Implicit_Packing`
  * `pragma Overriding_Renamings`
  * `pragma Use_VADS_Size`

2.144 Pragma Profile_Warnings

Syntax:

```
pragma Profile_Warnings (Ravenscar | Restricted | Rational);
```

This is an implementation-defined pragma that is similar in effect to `pragma Profile` except that instead of generating `Restrictions` pragmas, it generates `Restriction_Warnings` pragmas. The result is that violations of the profile generate warning messages instead of error messages.

2.145 Pragma Program_Exit

Syntax:

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

For the semantics of this pragma, see the entry for aspect `Program_Exit` in the SPARK 2014 Reference Manual, section 6.1.10.

2.146 Pragma Propagate_Exceptions

Syntax:

```
pragma Propagate_Exceptions;
```

This pragma is now obsolete and, other than generating a warning if warnings on obsolescent features are enabled, is ignored. It is retained for compatibility purposes. It used to be used in connection with optimization of a now-obsolete mechanism for implementation of exceptions.

2.147 Pragma Provide_Shift_Operators

Syntax:

```
pragma Provide_Shift_Operators (integer_first_subtype_LOCAL_NAME);
```

This pragma can be applied to a first subtype local name that specifies either an unsigned or signed type. It has the effect of providing the five shift operators (Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left and Rotate_Right) for the given type. It is similar to including the function declarations for these five operators, together with the pragma Import (Intrinsic, . . .) statements.

2.148 Pragma Psect_Object

Syntax:

```
pragma Psect_Object (
     [Internal =>] LOCAL_NAME,
  [, [External =>] EXTERNAL_SYMBOL]
  [, [Size     =>] EXTERNAL_SYMBOL]);

EXTERNAL_SYMBOL ::=
  IDENTIFIER
| static_string_EXPRESSION
```

This pragma is identical in effect to pragma `Common_Object`.

2.149 Pragma Pure_Function

Syntax:

```
pragma Pure_Function ([Entity =>] function_LOCAL_NAME);
```

This pragma appears in the same declarative part as a function declaration (or a set of function declarations if more than one overloaded declaration exists, in which case the pragma applies to all entities). It specifies that the function `Entity` is to be considered pure for the purposes of code generation. This means that the compiler can assume that there are no side effects, and in particular that two identical calls produce the same result in the same context. It also means that the function can be used in an address clause.

Note that, quite deliberately, there are no static checks to try to ensure that this promise is met, so `Pure_Function` can be used with functions that are conceptually pure, even if they do modify global variables. For example, a square root function that is instrumented to count the number of times it is called is still conceptually pure, and can still be optimized, even though it modifies a global variable (the count). Memo functions are another example (where a table of previous calls is kept and consulted to avoid re-computation).

Note also that the normal rules excluding optimization of subprograms in pure units (when parameter types are descended from System.Address, or when the full view of a parameter type is limited), do not apply for the Pure_Function case. If you explicitly specify Pure_Function, the compiler may optimize away calls with identical arguments, and if that results in unexpected behavior, the proper action is not to use the pragma for subprograms that are not (conceptually) pure.

Note: Most functions in a `Pure` package are automatically pure, and there is no need to use pragma `Pure_Function` for such functions. One exception is any function that has at least one formal of type `System.Address` or a type derived from it. Such functions are not considered pure by default, since the compiler assumes that the `Address` parameter may be functioning as a pointer and that the referenced data may change even if the address value does not. Similarly, imported functions are not considered to be pure by default, since there is no way of checking that they are in fact pure. The use of pragma `Pure_Function` for such a function will override these default assumption, and cause the compiler to treat a designated subprogram as pure in these cases.

Note: If pragma `Pure_Function` is applied to a renamed function, it applies to the underlying renamed function. This can be used to disambiguate cases of overloading where some but not all functions in a set of overloaded functions are to be designated as pure.

If pragma `Pure_Function` is applied to a library-level function, the function is also considered pure from an optimization point of view, but the unit is not a Pure unit in the categorization sense. So for example, a function thus marked is free to `with` non-pure units.

2.150 Pragma Rational

Syntax:

```
pragma Rational;
```

This pragma is considered obsolescent, but is retained for compatibility purposes. It is equivalent to:

```
pragma Profile (Rational);
```

2.151 Pragma Ravenscar

Syntax:

```
pragma Ravenscar;
```

This pragma is considered obsolescent, but is retained for compatibility purposes. It is equivalent to:

```
pragma Profile (Ravenscar);
```

which is the preferred method of setting the `Ravenscar` profile.

2.152 Pragma Refined_Depends

Syntax:

```
pragma Refined_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 `Refined_Depends` in the SPARK 2014 Reference Manual, section 6.1.5.

2.153 Pragma Refined_Global

Syntax:

```
pragma Refined_Global (GLOBAL_SPECIFICATION);

GLOBAL_SPECIFICATION ::=
     null
  | (GLOBAL_LIST)
  | (MODED_GLOBAL_LIST {, MODED_GLOBAL_LIST})

MODED_GLOBAL_LIST ::= MODE_SELECTOR => GLOBAL_LIST

MODE_SELECTOR ::= In_Out | Input | Output | Proof_In
GLOBAL_LIST   ::= GLOBAL_ITEM | (GLOBAL_ITEM {, GLOBAL_ITEM})
GLOBAL_ITEM   ::= NAME
```

For the semantics of this pragma, see the entry for aspect `Refined_Global` in the SPARK 2014 Reference Manual, section 6.1.4.

2.154 Pragma Refined_Post

Syntax:

```
pragma Refined_Post (boolean_EXPRESSION);
```

For the semantics of this pragma, see the entry for aspect `Refined_Post` in the SPARK 2014 Reference Manual, section 7.2.7.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.155 Pragma Refined_State

Syntax:

```
pragma Refined_State (REFINEMENT_LIST);

REFINEMENT_LIST ::=
  (REFINEMENT_CLAUSE {, REFINEMENT_CLAUSE})

REFINEMENT_CLAUSE ::= state_NAME => CONSTITUENT_LIST

CONSTITUENT_LIST ::=
```

```
       null
     |  CONSTITUENT
     | (CONSTITUENT {, CONSTITUENT})

CONSTITUENT ::= object_NAME | state_NAME
```

For the semantics of this pragma, see the entry for aspect `Refined_State` in the SPARK 2014 Reference Manual, section 7.2.2.

2.156 Pragma Relative_Deadline

Syntax:

```
pragma Relative_Deadline (time_span_EXPRESSION);
```

This pragma is standard in Ada 2005, but is available in all earlier versions of Ada as an implementation-defined pragma. See Ada 2012 Reference Manual for details.

2.157 Pragma Remote_Access_Type

Syntax:

```
pragma Remote_Access_Type ([Entity =>] formal_access_type_LOCAL_NAME);
```

This pragma appears in the formal part of a generic declaration. It specifies an exception to the RM rule from E.2.2(17/2), which forbids the use of a remote access to class-wide type as actual for a formal access type.

When this pragma applies to a formal access type `Entity`, that type is treated as a remote access to class-wide type in the generic. It must be a formal general access type, and its designated type must be the class-wide type of a formal tagged limited private type from the same generic declaration.

In the generic unit, the formal type is subject to all restrictions pertaining to remote access to class-wide types. At instantiation, the actual type must be a remote access to class-wide type.

2.158 Pragma Rename_Pragma

Syntax:

```
pragma Rename_Pragma (
         [New_Name =>] IDENTIFIER,
         [Renamed  =>] pragma_IDENTIFIER);
```

This pragma provides a mechanism for supplying new names for existing pragmas. The `New_Name` identifier can subsequently be used as a synonym for the Renamed pragma. For example, suppose you have code that was originally developed on a compiler that supports Inline_Only as an implementation defined pragma. And suppose the semantics of pragma Inline_Only are identical to (or at least very similar to) the GNAT implementation defined pragma Inline_Always. You could globally replace Inline_Only with Inline_Always.

However, to avoid that source modification, you could instead add a configuration pragma:

```
pragma Rename_Pragma (
         New_Name => Inline_Only,
```

```
                Renamed  => Inline_Always);
```

Then GNAT will treat "pragma Inline_Only ..." as if you had written "pragma Inline_Always ...".

Pragma Inline_Only will not necessarily mean the same thing as the other Ada compiler; it's up to you to make sure the semantics are close enough.

2.159 Pragma Restricted_Run_Time

Syntax:

```
pragma Restricted_Run_Time;
```

This pragma is considered obsolescent, but is retained for compatibility purposes. It is equivalent to:

```
pragma Profile (Restricted);
```

which is the preferred method of setting the restricted run time profile.

2.160 Pragma Restriction_Warnings

Syntax:

```
pragma Restriction_Warnings
  (restriction_IDENTIFIER {, restriction_IDENTIFIER});
```

This pragma allows a series of restriction identifiers to be specified (the list of allowed identifiers is the same as for pragma `Restrictions`). For each of these identifiers the compiler checks for violations of the restriction, but generates a warning message rather than an error message if the restriction is violated.

One use of this is in situations where you want to know about violations of a restriction, but you want to ignore some of these violations. Consider this example, where you want to set Ada_95 mode and enable style checks, but you want to know about any other use of implementation pragmas:

```
pragma Restriction_Warnings (No_Implementation_Pragmas);
pragma Warnings (Off, "violation of No_Implementation_Pragmas");
pragma Ada_95;
pragma Style_Checks ("2bfhkM160");
pragma Warnings (On, "violation of No_Implementation_Pragmas");
```

By including the above lines in a configuration pragmas file, the Ada_95 and Style_Checks pragmas are accepted without generating a warning, but any other use of implementation defined pragmas will cause a warning to be generated.

2.161 Pragma Reviewable

Syntax:

```
pragma Reviewable;
```

This pragma is an RM-defined standard pragma, but has no effect on the program being compiled, or on the code generated for the program.

To obtain the required output specified in RM H.3.1, the compiler must be run with various special switches as follows:

* 'Where compiler-generated run-time checks remain'

  The switch '-gnatGL' may be used to list the expanded code in pseudo-Ada form. Runtime checks show up in the listing either as explicit checks or operators marked with {} to indicate a check is present.

* 'An identification of known exceptions at compile time'

  If the program is compiled with '-gnatwa', the compiler warning messages will indicate all cases where the compiler detects that an exception is certain to occur at run time.

* 'Possible reads of uninitialized variables'

  The compiler warns of many such cases, but its output is incomplete.

A supplemental static analysis tool may be used to obtain a comprehensive list of all possible points at which uninitialized data may be read.

* 'Where run-time support routines are implicitly invoked'

  In the output from '-gnatGL', run-time calls are explicitly listed as calls to the relevant run-time routine.

* 'Object code listing'

  This may be obtained either by using the '-S' switch, or the objdump utility.

* 'Constructs known to be erroneous at compile time'

  These are identified by warnings issued by the compiler (use '-gnatwa').

* 'Stack usage information'

  Static stack usage data (maximum per-subprogram) can be obtained via the '-fstack-usage' switch to the compiler. Dynamic stack usage data (per task) can be obtained via the '-u' switch to gnatbind

* 'Object code listing of entire partition'

  This can be obtained by compiling the partition with '-S', or by applying objdump to all the object files that are part of the partition.

* 'A description of the run-time model'

  The full sources of the run-time are available, and the documentation of these routines describes how these run-time routines interface to the underlying operating system facilities.

* 'Control and data-flow information'

A supplemental static analysis tool may be used to obtain complete control and data-flow information, as well as comprehensive messages identifying possible problems based on this information.

2.162 Pragma Secondary_Stack_Size

Syntax:

```
pragma Secondary_Stack_Size (integer_EXPRESSION);
```

This pragma appears within the task definition of a single task declaration or a task type declaration (like pragma `Storage_Size`) and applies to all task objects of that type. The argument specifies the size of the secondary stack to be used by these task objects, and must be of an integer type. The secondary stack is used to handle functions that return a variable-sized result, for example a function returning an unconstrained String.

Note this pragma only applies to targets using fixed secondary stacks, like VxWorks 653 and bare board targets, where a fixed block for the secondary stack is allocated from the primary stack of the task. By default, these targets assign a percentage of the primary stack for the secondary stack, as defined by `System.Parameter.Sec_Stack_Percentage`. With this pragma, an `integer_EXPRESSION` of bytes is assigned from the primary stack instead.

For most targets, the pragma does not apply as the secondary stack grows on demand: allocated as a chain of blocks in the heap. The default size of these blocks can be modified via the `-D` binder option as described in *GNAT User's Guide*.

Note that no check is made to see if the secondary stack can fit inside the primary stack.

Note the pragma cannot appear when the restriction `No_Secondary_Stack` is in effect.

2.163 Pragma Share_Generic

Syntax:

```
pragma Share_Generic (GNAME {, GNAME});

GNAME ::= generic_unit_NAME | generic_instance_NAME
```

This pragma is provided for compatibility with Dec Ada 83. It has no effect in GNAT (which does not implement shared generics), other than to check that the given names are all names of generic units or generic instances.

2.164 Pragma Shared

This pragma is provided for compatibility with Ada 83. The syntax and semantics are identical to pragma Atomic.

2.165 Pragma Short_Circuit_And_Or

Syntax:

```
pragma Short_Circuit_And_Or;
```

This configuration pragma causes the predefined AND and OR operators of type Standard.Boolean to have short-circuit semantics. That is, they behave like AND THEN and OR ELSE; the right-hand side is not evaluated if the left-hand side determines the result. This may be useful in the context of certification protocols requiring the use of short-circuited logical operators.

There is no requirement that all units in a partition use this option. However, mixing of short-circuit and non-short-circuit semantics can be confusing. Therefore, the recommended use is to put the pragma in a configuration file that applies to the whole program. Alternatively, if you have a legacy library that should not use this pragma, you can put it in a separate library project that does not use the pragma. In any case, fine-grained mixing of the different semantics is not recommended. If pragma `Short_Circuit_And_Or` is specified, then it is illegal to rename the predefined Boolean AND and OR, or to pass them to generic formal functions; this corresponds to the fact that AND THEN and OR ELSE cannot be renamed nor passed as generic formal functions.

Note that this pragma has no effect on other logical operators – predefined operators of modular types, array-of-boolean types and types derived from Standard.Boolean, nor user-defined operators.

See also the pragma `Unevaluated_Use_Of_Old` and the restriction `No_Direct_Boolean_Operators`, which may be useful in conjunction with `Short_Circuit_And_Or`.

2.166 Pragma Short_Descriptors

Syntax:

```
pragma Short_Descriptors;
```

This pragma is provided for compatibility with other Ada implementations. It is recognized but ignored by all current versions of GNAT.

2.167 Pragma Side_Effects

Syntax:

```
pragma Side_Effects [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Side_Effects` in the SPARK Reference Manual, section 6.1.12.

2.168 Pragma Simple_Storage_Pool_Type

Syntax:

```
pragma Simple_Storage_Pool_Type (type_LOCAL_NAME);
```

A type can be established as a 'simple storage pool type' by applying the representation pragma `Simple_Storage_Pool_Type` to the type. A type named in the pragma must be a library-level immutably limited record type or limited tagged type declared immediately within a package declaration. The type can also be a limited private type whose full type is allowed as a simple storage pool type.

For a simple storage pool type `SSP`, nonabstract primitive subprograms `Allocate`, `Deallocate`, and `Storage_Size` can be declared that are subtype conformant with the following subprogram declarations:

```
procedure Allocate
  (Pool                     : in out SSP;
   Storage_Address          : out System.Address;
   Size_In_Storage_Elements : System.Storage_Elements.Storage_Count;
   Alignment                : System.Storage_Elements.Storage_Count);

procedure Deallocate
  (Pool : in out SSP;
   Storage_Address          : System.Address;
   Size_In_Storage_Elements : System.Storage_Elements.Storage_Count;
   Alignment                : System.Storage_Elements.Storage_Count);
```

```
function Storage_Size (Pool : SSP)
  return System.Storage_Elements.Storage_Count;
```

Procedure `Allocate` must be declared, whereas `Deallocate` and `Storage_Size` are optional. If `Deallocate` is not declared, then applying an unchecked deallocation has no effect other than to set its actual parameter to null. If `Storage_Size` is not declared, then the `Storage_Size` attribute applied to an access type associated with a pool object of type SSP returns zero. Additional operations can be declared for a simple storage pool type (such as for supporting a mark/release storage-management discipline).

An object of a simple storage pool type can be associated with an access type by specifying the attribute [Simple_Storage_Pool], page 136. For example:

```
My_Pool : My_Simple_Storage_Pool_Type;

type Acc is access My_Data_Type;

for Acc'Simple_Storage_Pool use My_Pool;
```

See attribute [Simple_Storage_Pool], page 136, for further details.

2.169 Pragma Source_File_Name

Syntax:

```
pragma Source_File_Name (
  [Unit_Name     =>] unit_NAME,
  Spec_File_Name =>  STRING_LITERAL,
  [Index => INTEGER_LITERAL]);

pragma Source_File_Name (
  [Unit_Name     =>] unit_NAME,
  Body_File_Name =>  STRING_LITERAL,
  [Index => INTEGER_LITERAL]);
```

Use this to override the normal naming convention. It is a configuration pragma, and so has the usual applicability of configuration pragmas (i.e., it applies to either an entire partition, or to all units in a compilation, or to a single unit, depending on how it is used). `unit_name` is mapped to `file_name_literal`. The identifier for the second argument is required, and indicates whether this is the file name for the spec or for the body.

The optional Index argument should be used when a file contains multiple units, and when you do not want to use `gnatchop` to separate then into multiple files (which is the recommended procedure to limit the number of recompilations that are needed when some sources change). For instance, if the source file `source.ada` contains

```
package B is
...
end B;

with B;
procedure A is
begin
```

```
     ..
  end A;
```

you could use the following configuration pragmas:

```
pragma Source_File_Name
  (B, Spec_File_Name => "source.ada", Index => 1);
pragma Source_File_Name
  (A, Body_File_Name => "source.ada", Index => 2);
```

Note that the `gnatname` utility can also be used to generate those configuration pragmas.

Another form of the `Source_File_Name` pragma allows the specification of patterns defining alternative file naming schemes to apply to all files.

```
pragma Source_File_Name
  (  [Spec_File_Name  =>] STRING_LITERAL
   [,[Casing          =>] CASING_SPEC]
   [,[Dot_Replacement =>] STRING_LITERAL]);

pragma Source_File_Name
  (  [Body_File_Name  =>] STRING_LITERAL
   [,[Casing          =>] CASING_SPEC]
   [,[Dot_Replacement =>] STRING_LITERAL]);

pragma Source_File_Name
  (  [Subunit_File_Name =>] STRING_LITERAL
   [,[Casing            =>] CASING_SPEC]
   [,[Dot_Replacement   =>] STRING_LITERAL]);

CASING_SPEC ::= Lowercase | Uppercase | Mixedcase
```

The first argument is a pattern that contains a single asterisk indicating the point at which the unit name is to be inserted in the pattern string to form the file name. The second argument is optional. If present it specifies the casing of the unit name in the resulting file name string. The default is lower case. Finally the third argument allows for systematic replacement of any dots in the unit name by the specified string literal.

Note that Source_File_Name pragmas should not be used if you are using project files. The reason for this rule is that the project manager is not aware of these pragmas, and so other tools that use the project file would not be aware of the intended naming conventions. If you are using project files, file naming is controlled by Source_File_Name_Project pragmas, which are usually supplied automatically by the project manager. A pragma Source_File_Name cannot appear after a [Pragma Source_File_Name_Project], page 86.

For more details on the use of the `Source_File_Name` pragma, see the sections on *Using Other File Names* and *Alternative File Naming Schemes* in the *GNAT User's Guide*.

2.170 Pragma Source_File_Name_Project

This pragma has the same syntax and semantics as pragma Source_File_Name. It is only allowed as a stand-alone configuration pragma. It cannot appear after a [Pragma Source_File_Name], page 85, and most importantly, once pragma Source_File_Name_Project appears, no further Source_File_Name pragmas are allowed.

The intention is that Source_File_Name_Project pragmas are always generated by the Project Manager in a manner consistent with the naming specified in a project file, and when naming is controlled in this manner, it is not permissible to attempt to modify this naming scheme using Source_File_Name or Source_File_Name_Project pragmas (which would not be known to the project manager).

2.171 Pragma Source_Reference

Syntax:

```
pragma Source_Reference (INTEGER_LITERAL, STRING_LITERAL);
```

This pragma must appear as the first line of a source file. `integer_literal` is the logical line number of the line following the pragma line (for use in error messages and debugging information). `string_literal` is a static string constant that specifies the file name to be used in error messages and debugging information. This is most notably used for the output of `gnatchop` with the '-r' switch, to make sure that the original unchopped source file is the one referred to.

The second argument must be a string literal, it cannot be a static string expression other than a string literal. This is because its value is needed for error messages issued by all phases of the compiler.

2.172 Pragma SPARK_Mode

Syntax:

```
pragma SPARK_Mode [(On | Off)] ;
```

In general a program can have some parts that are in SPARK 2014 (and follow all the rules in the SPARK Reference Manual), and some parts that are full Ada 2012.

The SPARK_Mode pragma is used to identify which parts are in SPARK 2014 (by default programs are in full Ada). The SPARK_Mode pragma can be used in the following places:

* As a configuration pragma, in which case it sets the default mode for all units compiled with this pragma.
* Immediately following a library-level subprogram spec
* Immediately within a library-level package body
* Immediately following the `private` keyword of a library-level package spec
* Immediately following the `begin` keyword of a library-level package body
* Immediately within a library-level subprogram body

Normally a subprogram or package spec/body inherits the current mode that is active at the point it is declared. But this can be overridden by pragma within the spec or body as above.

The basic consistency rule is that you can't turn SPARK_Mode back `On`, once you have explicitly (with a pragma) turned if `Off`. So the following rules apply:

If a subprogram spec has SPARK_Mode `Off`, then the body must also have SPARK_Mode `Off`.

For a package, we have four parts:

* the package public declarations

* the package private part
* the body of the package
* the elaboration code after `begin`

For a package, the rule is that if you explicitly turn SPARK_Mode `Off` for any part, then all the following parts must have SPARK_Mode `Off`. Note that this may require repeating a pragma SPARK_Mode (`Off`) in the body. For example, if we have a configuration pragma SPARK_Mode (`On`) that turns the mode on by default everywhere, and one particular package spec has pragma SPARK_Mode (`Off`), then that pragma will need to be repeated in the package body.

2.173 Pragma Static_Elaboration_Desired

Syntax:

```
pragma Static_Elaboration_Desired;
```

This pragma is used to indicate that the compiler should attempt to initialize statically the objects declared in the library unit to which the pragma applies, when these objects are initialized (explicitly or implicitly) by an aggregate. In the absence of this pragma, aggregates in object declarations are expanded into assignments and loops, even when the aggregate components are static constants. When the aggregate is present the compiler builds a static expression that requires no run-time code, so that the initialized object can be placed in read-only data space. If the components are not static, or the aggregate has more that 100 components, the compiler emits a warning that the pragma cannot be obeyed. (See also the restriction No_Implicit_Loops, which supports static construction of larger aggregates with static components that include an others choice.)

2.174 Pragma Stream_Convert

Syntax:

```
pragma Stream_Convert (
  [Entity =>] type_LOCAL_NAME,
  [Read   =>] function_NAME,
  [Write  =>] function_NAME);
```

This pragma provides an efficient way of providing user-defined stream attributes. Not only is it simpler to use than specifying the attributes directly, but more importantly, it allows the specification to be made in such a way that the predefined unit Ada.Streams is not loaded unless it is actually needed (i.e. unless the stream attributes are actually used); the use of the Stream_Convert pragma adds no overhead at all, unless the stream attributes are actually used on the designated type.

The first argument specifies the type for which stream functions are provided. The second parameter provides a function used to read values of this type. It must name a function whose argument type may be any subtype, and whose returned type must be the type given as the first argument to the pragma.

The meaning of the `Read` parameter is that if a stream attribute directly or indirectly specifies reading of the type given as the first parameter, then a value of the type given as the argument to the Read function is read from the stream, and then the Read function is used to convert this to the required target type.

Similarly the `Write` parameter specifies how to treat write attributes that directly or indirectly apply to the type given as the first parameter. It must have an input parameter of the type specified by the first parameter, and the return type must be the same as the input type of the Read function. The effect is to first call the Write function to convert to the given stream type, and then write the result type to the stream.

The Read and Write functions must not be overloaded subprograms. If necessary renamings can be supplied to meet this requirement. The usage of this attribute is best illustrated by a simple example, taken from the GNAT implementation of package Ada.Strings.Unbounded:

```
function To_Unbounded (S : String) return Unbounded_String
  renames To_Unbounded_String;

pragma Stream_Convert
  (Unbounded_String, To_Unbounded, To_String);
```

The specifications of the referenced functions, as given in the Ada Reference Manual are:

```
function To_Unbounded_String (Source : String)
  return Unbounded_String;

function To_String (Source : Unbounded_String)
  return String;
```

The effect is that if the value of an unbounded string is written to a stream, then the representation of the item in the stream is in the same format that would be used for `Standard.String'Output`, and this same representation is expected when a value of this type is read from the stream. Note that the value written always includes the bounds, even for Unbounded_String'Write, since Unbounded_String is not an array type.

Note that the `Stream_Convert` pragma is not effective in the case of a derived type of a non-limited tagged type. If such a type is specified then the pragma is silently ignored, and the default implementation of the stream attributes is used instead.

2.175 Pragma Style_Checks

Syntax:

```
pragma Style_Checks (string_LITERAL | ALL_CHECKS |
                     On | Off [, LOCAL_NAME]);
```

This pragma is used in conjunction with compiler switches to control the built in style checking provided by GNAT. The compiler switches, if set, provide an initial setting for the switches, and this pragma may be used to modify these settings, or the settings may be provided entirely by the use of the pragma. This pragma can be used anywhere that a pragma is legal, including use as a configuration pragma (including use in the `gnat.adc` file).

The form with a string literal specifies which style options are to be activated. These are additive, so they apply in addition to any previously set style check options. The codes for the options are the same as those used in the '-gnaty' switch to 'gcc' or 'gnatmake'. For example the following two methods can be used to enable layout checking and to change the maximum nesting level value:

*

```
--  switch on layout checks
pragma Style_Checks ("l");
--  set the number of maximum allowed nesting levels to 15
pragma Style_Checks ("L15");
```

*

```
gcc -c -gnatyl -gnatyL15 ...
```

The string literal values can be cumulatively switched on and off by prefixing the value with `+` or `-`, where:

* `+` is equivalent to no prefix. It applies the check referenced by the literal value;
* `-` switches the referenced check off.

```
--  allow misaligned block by disabling layout check
pragma Style_Checks ("-l");
declare
    msg : constant String := "Hello";
begin
    Put_Line (msg);
    end;

--  enable the layout check again
pragma Style_Checks ("l");
declare
    msg : constant String := "Hello";
begin
    Put_Line (msg);
    end;
```

The code above contains two layout errors, however, only the last line is picked up by the compiler.

Similarly, the switches containing a numeric value can be applied in sequence. In the example below, the permitted nesting level is reduced in in the middle block and the compiler raises a warning on the highlighted line.

```
-- Permit 3 levels of nesting
pragma Style_Checks ("L3");

procedure Main is
begin
    if True then
      if True then
          null;
      end if;
    end if;
    --  Reduce permitted nesting levels to 2.
    --  Note that "+L2" and "L2" are equivalent.
    pragma Style_Checks ("+L2");
    if True then
      if True then
```

```
              null;
          end if;
        end if;
        --  Disable checking permitted nesting levels.
        --  Note that the number after "-L" is insignificant,
        --  "-L", "-L3" and "-Lx" are all equivalent.
        pragma Style_Checks ("-L3");
        if True then
          if True then
              null;
          end if;
        end if;
    end Main;
```

The form `ALL_CHECKS` activates all standard checks (its use is equivalent to the use of the `gnaty` switch with no options. See the *GNAT User's Guide* for details.)

Note: the behavior is slightly different in GNAT mode (`-gnatg` used). In this case, `ALL_CHECKS` implies the standard set of GNAT mode style check options (i.e. equivalent to `-gnatyg`).

The forms with `Off` and `On` can be used to temporarily disable style checks as shown in the following example:

```
pragma Style_Checks ("k"); -- requires keywords in lower case
pragma Style_Checks (Off); -- turn off style checks
NULL;                      -- this will not generate an error message
pragma Style_Checks (On);  -- turn style checks back on
NULL;                      -- this will generate an error message
```

Finally the two argument form is allowed only if the first argument is `On` or `Off`. The effect is to turn of semantic style checks for the specified entity, as shown in the following example:

```
pragma Style_Checks ("r"); -- require consistency of identifier casing
Arg : Integer;
Rf1 : Integer := ARG;      -- incorrect, wrong case
pragma Style_Checks (Off, Arg);
Rf2 : Integer := ARG;      -- OK, no error
```

2.176 Pragma Subprogram_Variant

Syntax:

```
pragma Subprogram_Variant (SUBPROGRAM_VARIANT_LIST);

SUBPROGRAM_VARIANT_LIST ::=
  STRUCTURAL_SUBPROGRAM_VARIANT_ITEM
| NUMERIC_SUBPROGRAM_VARIANT_ITEMS

NUMERIC_SUBPROGRAM_VARIANT_ITEMS ::=
  NUMERIC_SUBPROGRAM_VARIANT_ITEM {, NUMERIC_SUBPROGRAM_VARIANT_ITEM}

NUMERIC_SUBPROGRAM_VARIANT_ITEM ::=
```

```
    CHANGE_DIRECTION => EXPRESSION

  STRUCTURAL_SUBPROGRAM_VARIANT_ITEM ::=
    STRUCTURAL => EXPRESSION

  CHANGE_DIRECTION ::= Increases | Decreases
```

The `Subprogram_Variant` pragma is intended to be an exact replacement for the implementation-defined `Subprogram_Variant` aspect, and shares its restrictions and semantics.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.177 Pragma Subtitle

Syntax:

```
pragma Subtitle ([Subtitle =>] STRING_LITERAL);
```

This pragma is recognized for compatibility with other Ada compilers but is ignored by GNAT.

2.178 Pragma Suppress

Syntax:

```
pragma Suppress (Identifier [, [On =>] Name]);
```

This is a standard pragma, and supports all the check names required in the RM. It is included here because GNAT recognizes some additional check names that are implementation defined (as permitted by the RM):

* `Alignment_Check` can be used to suppress alignment checks on addresses used in address clauses. Note that `Alignment_Check` is suppressed by default on non-strict alignment machines (such as the x86).
* `Atomic_Synchronization` can be used to suppress the special memory synchronization instructions that are normally generated for access to `Atomic` variables to ensure correct synchronization between tasks that use such variables for synchronization purposes.
* `Duplicated_Tag_Check` Can be used to suppress the check that is generated for a duplicated tag value when a tagged type is declared.
* `Container_Checks` Can be used to suppress all checks within Ada.Containers and instances of its children, including Tampering_Check.
* `Tampering_Check` Can be used to suppress tampering check in the containers.
* `Predicate_Check` can be used to control whether predicate checks are active. It is applicable only to predicates for which the policy is `Check`. Unlike `Assertion_Policy`, which determines if a given predicate is ignored or checked for the whole program, the use of `Suppress` and `Unsuppress` with this check name allows a given predicate to be turned on and off at specific points in the program.
* `Validity_Check` can be used specifically to control validity checks. If `Suppress` is used to suppress validity checks, then no validity checks are performed, including those specified by the appropriate compiler switch or the `Validity_Checks` pragma.

* Additional check names previously introduced by use of the `Check_Name` pragma are also allowed.

Note that pragma Suppress gives the compiler permission to omit checks, but does not require the compiler to omit checks. The compiler will generate checks if they are essentially free, even when they are suppressed. In particular, if the compiler can prove that a certain check will necessarily fail, it will generate code to do an unconditional 'raise', even if checks are suppressed. The compiler warns in this case.

Of course, run-time checks are omitted whenever the compiler can prove that they will not fail, whether or not checks are suppressed.

2.179 Pragma Suppress_All

Syntax:

```
pragma Suppress_All;
```

This pragma can appear anywhere within a unit. The effect is to apply `Suppress (All_Checks)` to the unit in which it appears. This pragma is implemented for compatibility with DEC Ada 83 usage where it appears at the end of a unit, and for compatibility with Rational Ada, where it appears as a program unit pragma. The use of the standard Ada pragma `Suppress (All_Checks)` as a normal configuration pragma is the preferred usage in GNAT.

2.180 Pragma Suppress_Debug_Info

Syntax:

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

This pragma can be used to suppress generation of debug information for the specified entity. It is intended primarily for use in debugging the debugger, and navigating around debugger problems.

2.181 Pragma Suppress_Exception_Locations

Syntax:

```
pragma Suppress_Exception_Locations;
```

In normal mode, a raise statement for an exception by default generates an exception message giving the file name and line number for the location of the raise. This is useful for debugging and logging purposes, but this entails extra space for the strings for the messages. The configuration pragma `Suppress_Exception_Locations` can be used to suppress the generation of these strings, with the result that space is saved, but the exception message for such raises is null. This configuration pragma may appear in a global configuration pragma file, or in a specific unit as usual. It is not required that this pragma be used consistently within a partition, so it is fine to have some units within a partition compiled with this pragma and others compiled in normal mode without it.

2.182 Pragma Suppress_Initialization

Syntax:

```
pragma Suppress_Initialization ([Entity =>] variable_or_subtype_LOCAL_NAME);
```

Here variable_or_subtype_LOCAL_NAME is the name introduced by a type declaration or subtype declaration or the name of a variable introduced by an object declaration.

In the case of a type or subtype this pragma suppresses any implicit or explicit initialization for all variables of the given type or subtype, including initialization resulting from the use of pragmas Normalize_Scalars or Initialize_Scalars.

This is considered a representation item, so it cannot be given after the type is frozen. It applies to all subsequent object declarations, and also any allocator that creates objects of the type.

If the pragma is given for the first subtype, then it is considered to apply to the base type and all its subtypes. If the pragma is given for other than a first subtype, then it applies only to the given subtype. The pragma may not be given after the type is frozen.

Note that this includes eliminating initialization of discriminants for discriminated types, and tags for tagged types. In these cases, you will have to use some non-portable mechanism (e.g. address overlays or unchecked conversion) to achieve required initialization of these fields before accessing any object of the corresponding type.

For the variable case, implicit initialization for the named variable is suppressed, just as though its subtype had been given in a pragma Suppress_Initialization, as described above.

2.183 Pragma Task_Name

Syntax

```
pragma Task_Name (string_EXPRESSION);
```

This pragma appears within a task definition (like pragma `Priority`) and applies to the task in which it appears. The argument must be of type String, and provides a name to be used for the task instance when the task is created. Note that this expression is not required to be static, and in particular, it can contain references to task discriminants. This facility can be used to provide different names for different tasks as they are created, as illustrated in the example below.

The task name is recorded internally in the run-time structures and is accessible to tools like the debugger. In addition the routine `Ada.Task_Identification.Image` will return this string, with a unique task address appended.

```
--  Example of the use of pragma Task_Name

with Ada.Task_Identification;
use Ada.Task_Identification;
with Text_IO; use Text_IO;
procedure t3 is

   type Astring is access String;

   task type Task_Typ (Name : access String) is
```

```
      pragma Task_Name (Name.all);
   end Task_Typ;

   task body Task_Typ is
      Nam : constant String := Image (Current_Task);
   begin
      Put_Line ("-->" & Nam (1 .. 14) & "<--");
   end Task_Typ;

   type Ptr_Task is access Task_Typ;
   Task_Var : Ptr_Task;

begin
   Task_Var :=
     new Task_Typ (new String'("This is task 1"));
   Task_Var :=
     new Task_Typ (new String'("This is task 2"));
end;
```

2.184 Pragma Task_Storage

Syntax:

```
pragma Task_Storage (
  [Task_Type =>] LOCAL_NAME,
  [Top_Guard =>] static_integer_EXPRESSION);
```

This pragma specifies the length of the guard area for tasks. The guard area is an additional storage area allocated to a task. A value of zero means that either no guard area is created or a minimal guard area is created, depending on the target. This pragma can appear anywhere a `Storage_Size` attribute definition clause is allowed for a task type.

2.185 Pragma Test_Case

Syntax:

```
pragma Test_Case (
   [Name     =>] static_string_Expression
  ,[Mode     =>] (Nominal | Robustness)
 [, Requires =>  Boolean_Expression]
 [, Ensures  =>  Boolean_Expression]);
```

The `Test_Case` pragma allows defining fine-grain specifications for use by testing tools. The compiler checks the validity of the `Test_Case` pragma, but its presence does not lead to any modification of the code generated by the compiler.

`Test_Case` pragmas may only appear immediately following the (separate) declaration of a subprogram in a package declaration, inside a package spec unit. Only other pragmas may intervene (that is appear between the subprogram declaration and a test case).

The compiler checks that boolean expressions given in `Requires` and `Ensures` are valid, where the rules for `Requires` are the same as the rule for an expression in `Precondition` and the rules for `Ensures` are the same as the rule for an expression in `Postcondition`. In particular, attributes `'Old` and `'Result` can only be used within the `Ensures` expression. The following is an example of use within a package spec:

```
package Math_Functions is
   ...
   function Sqrt (Arg : Float) return Float;
   pragma Test_Case (Name     => "Test 1",
                     Mode     => Nominal,
                     Requires => Arg < 10000.0,
                     Ensures  => Sqrt'Result < 10.0);
   ...
end Math_Functions;
```

The meaning of a test case is that there is at least one context where `Requires` holds such that, if the associated subprogram is executed in that context, then `Ensures` holds when the subprogram returns. Mode `Nominal` indicates that the input context should also satisfy the precondition of the subprogram, and the output context should also satisfy its postcondition. Mode `Robustness` indicates that the precondition and postcondition of the subprogram should be ignored for this test case.

2.186 Pragma Thread_Local_Storage

Syntax:

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

This pragma specifies that the specified entity, which must be a variable declared in a library-level package, is to be marked as "Thread Local Storage" (`TLS`). On systems supporting this (which include Windows, Solaris, GNU/Linux, and VxWorks), this causes each thread (and hence each Ada task) to see a distinct copy of the variable.

The variable must not have default initialization, and if there is an explicit initialization, it must be either `null` for an access variable, a static expression for a scalar variable, or a fully static aggregate for a composite type, that is to say, an aggregate all of whose components are static, and which does not include packed or discriminated components.

This provides a low-level mechanism similar to that provided by the `Ada.Task_Attributes` package, but much more efficient and is also useful in writing interface code that will interact with foreign threads.

If this pragma is used on a system where `TLS` is not supported, then an error message will be generated and the program will be rejected.

2.187 Pragma Time_Slice

Syntax:

```
pragma Time_Slice (static_duration_EXPRESSION);
```

For implementations of GNAT on operating systems where it is possible to supply a time slice value, this pragma may be used for this purpose. It is ignored if it is used in a system that does not allow this control, or if it appears in other than the main program unit.

2.188 Pragma Title

Syntax:

```
pragma Title (TITLING_OPTION [, TITLING OPTION]);

TITLING_OPTION ::=
  [Title    =>] STRING_LITERAL,
| [Subtitle =>] STRING_LITERAL
```

Syntax checked but otherwise ignored by GNAT. This is a listing control pragma used in DEC Ada 83 implementations to provide a title and/or subtitle for the program listing. The program listing generated by GNAT does not have titles or subtitles.

Unlike other pragmas, the full flexibility of named notation is allowed for this pragma, i.e., the parameters may be given in any order if named notation is used, and named and positional notation can be mixed following the normal rules for procedure calls in Ada.

2.189 Pragma Type_Invariant

Syntax:

```
pragma Type_Invariant
  ([Entity =>] type_LOCAL_NAME,
   [Check  =>] EXPRESSION);
```

The `Type_Invariant` pragma is intended to be an exact replacement for the language-defined `Type_Invariant` aspect, and shares its restrictions and semantics. It differs from the language defined `Invariant` pragma in that it does not permit a string parameter, and it is controlled by the assertion identifier `Type_Invariant` rather than `Invariant`.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.190 Pragma Type_Invariant_Class

Syntax:

```
pragma Type_Invariant_Class
  ([Entity =>] type_LOCAL_NAME,
   [Check  =>] EXPRESSION);
```

The `Type_Invariant_Class` pragma is intended to be an exact replacement for the language-defined `Type_Invariant'Class` aspect, and shares its restrictions and semantics.

Note: This pragma is called `Type_Invariant_Class` rather than `Type_Invariant'Class` because the latter would not be strictly conforming to the allowed syntax for pragmas. The motivation for providing pragmas equivalent to the aspects is to allow a program to be written using the pragmas, and then compiled if necessary using an Ada compiler that does not recognize the pragmas or aspects, but is prepared to ignore the pragmas. The assertion policy that controls this pragma is `Type_Invariant'Class`, not `Type_Invariant_Class`.

This is an assertion kind pragma that can associate a set of its arguments with an assertion level. See SPARK 2014 Reference Manual, section 11.4.2.

2.191 Pragma Unchecked_Union

Syntax:

```
pragma Unchecked_Union (first_subtype_LOCAL_NAME);
```

This pragma is used to specify a representation of a record type that is equivalent to a C union. It was introduced as a GNAT implementation defined pragma in the GNAT Ada 95 mode. Ada 2005 includes an extended version of this pragma, making it language defined, and GNAT fully implements this extended version in all language modes (Ada 83, Ada 95, and Ada 2005). For full details, consult the Ada 2012 Reference Manual, section B.3.3.

2.192 Pragma Unevaluated_Use_Of_Old

Syntax:

```
pragma Unevaluated_Use_Of_Old (Error | Warn | Allow);
```

This pragma controls the processing of attributes Old and Loop_Entry. If either of these attributes is used in a potentially unevaluated expression (e.g. the then or else parts of an if expression), then normally this usage is considered illegal if the prefix of the attribute is other than an entity name. The language requires this behavior for Old, and GNAT copies the same rule for Loop_Entry.

The reason for this rule is that otherwise, we can have a situation where we save the Old value, and this results in an exception, even though we might not evaluate the attribute. Consider this example:

```
package UnevalOld is
   K : Character;
   procedure U (A : String; C : Boolean)  -- ERROR
     with Post => (if C then A(1)'Old = K else True);
end;
```

If procedure U is called with a string with a lower bound of 2, and C false, then an exception would be raised trying to evaluate A(1) on entry even though the value would not be actually used.

Although the rule guarantees against this possibility, it is sometimes too restrictive. For example if we know that the string has a lower bound of 1, then we will never raise an exception. The pragma `Unevaluated_Use_Of_Old` can be used to modify this behavior. If the argument is `Error` then an error is given (this is the default RM behavior). If the argument is `Warn` then the usage is allowed as legal but with a warning that an exception might be raised. If the argument is `Allow` then the usage is allowed as legal without generating a warning.

This pragma may appear as a configuration pragma, or in a declarative part or package specification. In the latter case it applies to uses up to the end of the corresponding statement sequence or sequence of package declarations.

2.193 Pragma User_Aspect_Definition

Syntax:

```
pragma User_Aspect_Definition
  (Identifier {, Identifier [(Identifier {, Identifier})]});
```

This configuration pragma defines a new aspect, making it available for subsequent use in a *User_Aspect* aspect specification. The first identifier is the name of the new aspect. Any subsequent arguments specify the names of other aspects. A subsequent name for which no parenthesized arguments are given shall denote either a Boolean-valued non-representation aspect or an aspect that has been defined by another *User_Aspect_Definition* pragma. A name for which one or more arguments are given shall be either *Annotate* or *Local_Restrictions* (and the arguments shall be appropriate for the named aspect).

This pragma, together with the *User_Aspect* aspect, provides a mechanism for avoiding textual duplication if some set of aspect specifications is needed in multiple places. This is somewhat analogous to how profiles allow avoiding duplication of *Restrictions* pragmas.

The visibility rules for an aspect defined by a *User_Aspect_Definition* pragma are the same as for a check name introduced by a *Check_Name* pragma. If multiple definitions are visible for some aspect at some point, then the definitions must agree. A predefined aspect cannot be redefined.

2.194 Pragma Unimplemented_Unit

Syntax:

```
pragma Unimplemented_Unit;
```

If this pragma occurs in a unit that is processed by the compiler, GNAT aborts with the message `xxx is not supported in this configuration`, where `xxx` is the name of the current compilation unit. This pragma is intended to allow the compiler to handle unimplemented library units in a clean manner.

The abort only happens if code is being generated. Thus you can use specs of unimplemented packages in syntax or semantic checking mode.

2.195 Pragma Universal_Aliasing

Syntax:

```
pragma Universal_Aliasing [([Entity =>] type_LOCAL_NAME)];
```

`type_LOCAL_NAME` must refer to a type declaration in the current declarative part. The effect is to inhibit strict type-based aliasing optimizations for the given type. For a detailed description of the strict type-based aliasing optimizations and the situations in which they need to be suppressed, see the section on `Optimization and Strict Aliasing` in the *GNAT User's Guide*.

2.196 Pragma Unmodified

Syntax:

```
pragma Unmodified (LOCAL_NAME {, LOCAL_NAME});
```

This pragma signals that the assignable entities (variables, `out` parameters, `in out` parameters) whose names are listed are deliberately not assigned in the current source unit. This suppresses warnings about the entities being referenced but not assigned, and in addition a warning will be generated if one of these entities is in fact assigned in the same unit as the pragma (or in the corresponding body, or one of its subunits).

This is particularly useful for clearly signaling that a particular parameter is not modified, even though the spec suggests that it might be.

For the variable case, warnings are never given for unreferenced variables whose name contains one of the substrings `DISCARD, DUMMY, IGNORE, JUNK, UNUSE, TMP, TEMP` in any casing. Such names are typically to be used in cases where such warnings are expected. Thus it is never necessary to use `pragma Unmodified` for such variables, though it is harmless to do so.

2.197 Pragma Unreferenced

Syntax:

```
pragma Unreferenced (LOCAL_NAME {, LOCAL_NAME});
pragma Unreferenced (library_unit_NAME {, library_unit_NAME});
```

This pragma signals that the entities whose names are listed are deliberately not referenced in the current source unit after the occurrence of the pragma. This suppresses warnings about the entities being unreferenced, and in addition a warning will be generated if one of these entities is in fact subsequently referenced in the same unit as the pragma (or in the corresponding body, or one of its subunits).

This is particularly useful for clearly signaling that a particular parameter is not referenced in some particular subprogram implementation and that this is deliberate. It can also be useful in the case of objects declared only for their initialization or finalization side effects.

If `LOCAL_NAME` identifies more than one matching homonym in the current scope, then the entity most recently declared is the one to which the pragma applies. Note that in the case of accept formals, the pragma Unreferenced may appear immediately after the keyword `do` which allows the indication of whether or not accept formals are referenced or not to be given individually for each accept statement.

The left hand side of an assignment does not count as a reference for the purpose of this pragma. Thus it is fine to assign to an entity for which pragma Unreferenced is given. However, use of an entity as an actual for an out parameter does count as a reference unless warnings for unread output parameters are enabled via `-gnatw.o`.

Note that if a warning is desired for all calls to a given subprogram, regardless of whether they occur in the same unit as the subprogram declaration, then this pragma should not be used (calls from another unit would not be flagged); pragma Obsolescent can be used instead for this purpose, see [Pragma Obsolescent], page 61.

The second form of pragma `Unreferenced` is used within a context clause. In this case the arguments must be unit names of units previously mentioned in `with` clauses (similar to the usage of pragma `Elaborate_All`). The effect is to suppress warnings about unreferenced units and unreferenced entities within these units.

For the variable case, warnings are never given for unreferenced variables whose name contains one of the substrings `DISCARD, DUMMY, IGNORE, JUNK, UNUSE, TMP, TEMP` in any casing. Such names are typically to be used in cases where such warnings are expected. Thus it is never necessary to use `pragma Unreferenced` for such variables, though it is harmless to do so.

2.198 Pragma Unreferenced_Objects

Syntax:

```
pragma Unreferenced_Objects (local_subtype_NAME {, local_subtype_NAME});
```

This pragma signals that for the types or subtypes whose names are listed, objects which are declared with one of these types or subtypes may not be referenced, and if no references appear, no warnings are given.

This is particularly useful for objects which are declared solely for their initialization and finalization effect. Such variables are sometimes referred to as RAII variables (Resource Acquisition Is Initialization). Using this pragma on the relevant type (most typically a limited controlled type), the compiler will automatically suppress unwanted warnings about these variables not being referenced.

2.199 Pragma Unreserve_All_Interrupts

Syntax:

```
pragma Unreserve_All_Interrupts;
```

Normally certain interrupts are reserved to the implementation. Any attempt to attach an interrupt causes Program_Error to be raised, as described in RM C.3.2(22). A typical example is the `SIGINT` interrupt used in many systems for a `Ctrl-C` interrupt. Normally this interrupt is reserved to the implementation, so that `Ctrl-C` can be used to interrupt execution.

If the pragma `Unreserve_All_Interrupts` appears anywhere in any unit in a program, then all such interrupts are unreserved. This allows the program to handle these interrupts, but disables their standard functions. For example, if this pragma is used, then pressing `Ctrl-C` will not automatically interrupt execution. However, a program can then handle the `SIGINT` interrupt as it chooses.

For a full list of the interrupts handled in a specific implementation, see the source code for the spec of `Ada.Interrupts.Names` in file `a-intnam.ads`. This is a target dependent file that contains the list of interrupts recognized for a given target. The documentation in this file also specifies what interrupts are affected by the use of the `Unreserve_All_Interrupts` pragma.

For a more general facility for controlling what interrupts can be handled, see pragma `Interrupt_State`, which subsumes the functionality of the `Unreserve_All_Interrupts` pragma.

2.200 Pragma Unsuppress

Syntax:

```
pragma Unsuppress (IDENTIFIER [, [On =>] NAME]);
```

This pragma undoes the effect of a previous pragma `Suppress`. If there is no corresponding pragma `Suppress` in effect, it has no effect. The range of the effect is the same as for pragma `Suppress`. The meaning of the arguments is identical to that used in pragma `Suppress`.

One important application is to ensure that checks are on in cases where code depends on the checks for its correct functioning, so that the code will compile correctly even if the compiler switches are set to suppress checks. For example, in a program that depends on external names of tagged types and wants to ensure that the duplicated tag check occurs even if all run-time checks are suppressed by a compiler switch, the following configuration pragma will ensure this test is not suppressed:

```
pragma Unsuppress (Duplicated_Tag_Check);
```

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

Note that in addition to the checks defined in the Ada RM, GNAT recognizes a number of implementation-defined check names. See the description of pragma `Suppress` for full details.

2.201 Pragma Unused

Syntax:

```
pragma Unused (LOCAL_NAME {, LOCAL_NAME});
```

This pragma signals that the assignable entities (variables, `out` parameters, and `in out` parameters) whose names are listed deliberately do not get assigned or referenced in the current source unit after the occurrence of the pragma in the current source unit. This suppresses warnings about the entities that are unreferenced and/or not assigned, and, in addition, a warning will be generated if one of these entities gets assigned or subsequently referenced in the same unit as the pragma (in the corresponding body or one of its subunits).

This is particularly useful for clearly signaling that a particular parameter is not modified or referenced, even though the spec suggests that it might be.

For the variable case, warnings are never given for unreferenced variables whose name contains one of the substrings `DISCARD, DUMMY, IGNORE, JUNK, UNUSE, TMP, TEMP` in any casing. Such names are typically to be used in cases where such warnings are expected. Thus it is never necessary to use `pragma Unused` for such variables, though it is harmless to do so.

2.202 Pragma Use_VADS_Size

Syntax:

```
pragma Use_VADS_Size;
```

This is a configuration pragma. In a unit to which it applies, any use of the 'Size attribute is automatically interpreted as a use of the 'VADS_Size attribute. Note that this may result in incorrect semantic processing of valid Ada 95 or Ada 2005 programs. This is intended to aid in the handling of existing code which depends on the interpretation of Size as implemented in the VADS compiler. See description of the VADS_Size attribute for further details.

2.203 Pragma Validity_Checks

Syntax:

```
pragma Validity_Checks (string_LITERAL | ALL_CHECKS | On | Off);
```

This pragma is used in conjunction with compiler switches to control the built-in validity checking provided by GNAT. The compiler switches, if set provide an initial setting for the switches, and this pragma may be used to modify these settings, or the settings may be provided entirely by the use of the pragma. This pragma can be used anywhere that a pragma is legal, including use as a configuration pragma (including use in the `gnat.adc` file).

The form with a string literal specifies which validity options are to be activated. The validity checks are first set to include only the default reference manual settings, and then a string of letters in the string specifies the exact set of options required. The form of this string is exactly as described for the '-gnatVx' compiler switch (see the GNAT User's Guide for details). For example the following two methods can be used to enable validity checking for mode `in` and `in out` subprogram parameters:

*
```
pragma Validity_Checks ("im");
```
*
```
$ gcc -c -gnatVim ...
```

The form ALL_CHECKS activates all standard checks (its use is equivalent to the use of the `gnatVa` switch).

The forms with `Off` and `On` can be used to temporarily disable validity checks as shown in the following example:

```
pragma Validity_Checks ("c"); -- validity checks for copies
pragma Validity_Checks (Off); -- turn off validity checks
A := B;                       -- B will not be validity checked
pragma Validity_Checks (On);  -- turn validity checks back on
A := C;                       -- C will be validity checked
```

2.204 Pragma Volatile

Syntax:

```
pragma Volatile (LOCAL_NAME);
```

This pragma is defined by the Ada Reference Manual, and the GNAT implementation is fully conformant with this definition. The reason it is mentioned in this section is that a pragma of the same name was supplied in some Ada 83 compilers, including DEC Ada 83. The Ada 95 / Ada 2005 implementation of pragma Volatile is upwards compatible with the implementation in DEC Ada 83.

2.205 Pragma Volatile_Full_Access

Syntax:

```
pragma Volatile_Full_Access (LOCAL_NAME);
```

This is similar in effect to pragma Volatile, except that any reference to the object is guaranteed to be done only with instructions that read or write all the bits of the object. Furthermore, if the object is of a composite type, then any reference to a subcomponent of the object is guaranteed to read and/or write all the bits of the object.

The intention is that this be suitable for use with memory-mapped I/O devices on some machines. Note that there are two important respects in which this is different from `pragma Atomic`. First a reference to a `Volatile_Full_Access` object is not a sequential action in the RM 9.10 sense and, therefore, does not create a synchronization point. Second, in the case of `pragma Atomic`, there is no guarantee that all the bits will be accessed if the reference is not to the whole object; the compiler is allowed (and generally will) access only part of the object in this case.

2.206 Pragma Volatile_Function

Syntax:

```
pragma Volatile_Function [ (static_boolean_EXPRESSION) ];
```

For the semantics of this pragma, see the entry for aspect `Volatile_Function` in the SPARK 2014 Reference Manual, section 7.1.2.

2.207 Pragma Warning_As_Error

Syntax:

```
pragma Warning_As_Error (static_string_EXPRESSION);
```

This configuration pragma allows the programmer to specify a set of warnings that will be treated as errors. Any warning that matches the pattern given by the pragma argument will be treated as an error. This gives more precise control than -gnatwe, which treats warnings as errors.

This pragma can apply to regular warnings (messages enabled by -gnatw) and to style warnings (messages that start with "(style)", enabled by -gnaty).

The pattern may contain asterisks, which match zero or more characters in the message. For example, you can use `pragma Warning_As_Error ("bits of*unused")` to treat the warning message `warning: 960 bits of "a" unused` as an error. All characters other than asterisk are treated as literal characters in the match. The match is case insensitive; for example XYZ matches xyz.

Note that the pattern matches if it occurs anywhere within the warning message string (it is not necessary to put an asterisk at the start and the end of the message, since this is implied).

Another possibility for the static_string_EXPRESSION which works whether or not error tags are enabled ('-gnatw.d') is to use a single '-gnatw' tag string, enclosed in brackets, as shown in the example below, to treat one category of warnings as errors. Note that if you want to treat multiple categories of warnings as errors, you can use multiple pragma Warning_As_Error.

The above use of patterns to match the message applies only to warning messages generated by the front end. This pragma can also be applied to warnings provided by the back end and mentioned in [Pragma Warnings], page 105. By using a single full '-Wxxx' switch in the pragma, such warnings can also be treated as errors.

The pragma can appear either in a global configuration pragma file (e.g. `gnat.adc`), or at the start of a file. Given a global configuration pragma file containing:

```
pragma Warning_As_Error ("[-gnatwj]");
```

which will treat all obsolescent feature warnings as errors, the following program compiles as shown (compile options here are '-gnatwa.d -gnatl -gnatj55').

```
    1. pragma Warning_As_Error ("*never assigned*");
```

```
      2. function Warnerr return String is
      3.    X : Integer;
            |
         >>> error: variable "X" is never read and
             never assigned [-gnatwv] [warning-as-error]

      4.    Y : Integer;
            |
         >>> warning: variable "Y" is assigned but
             never read [-gnatwu]

      5. begin
      6.    Y := 0;
      7.    return %ABC%;
                   |
         >>> error: use of "%" is an obsolescent
             feature (RM J.2(4)), use """ instead
             [-gnatwj] [warning-as-error]

      8. end;

 8 lines: No errors, 3 warnings (2 treated as errors)
```

Note that this pragma does not affect the set of warnings issued in any way, it merely changes the effect of a matching warning if one is produced as a result of other warnings options. As shown in this example, if the pragma results in a warning being treated as an error, the tag is changed from "warning:" to "error:" and the string "[warning-as-error]" is appended to the end of the message.

2.208 Pragma Warnings

Syntax:

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

DETAILS ::= On | Off
DETAILS ::= On | Off, local_NAME
DETAILS ::= static_string_EXPRESSION
DETAILS ::= On | Off, static_string_EXPRESSION

TOOL_NAME ::= GNAT | GNATprove

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

Note: in Ada 83 mode, a string literal may be used in place of a static string expression (which does not exist in Ada 83).

Note if the second argument of `DETAILS` is a `local_NAME` then the second form is always understood. If the intention is to use the fourth form, then you can write `NAME & ""` to force the interpretation as a 'static_string_EXPRESSION'.

Note: if the first argument is a valid `TOOL_NAME`, it will be interpreted that way. The use of the `TOOL_NAME` argument is relevant only to users of SPARK and GNATprove, see last part of this section for details.

Normally warnings are enabled, with the output being controlled by the command line switch. Warnings (`Off`) turns off generation of warnings until a Warnings (`On`) is encountered or the end of the current unit. If generation of warnings is turned off using this pragma, then some or all of the warning messages are suppressed, regardless of the setting of the command line switches.

The `Reason` parameter may optionally appear as the last argument in any of the forms of this pragma. It is intended purely for the purposes of documenting the reason for the `Warnings` pragma. The compiler will check that the argument is a static string but otherwise ignore this argument. Other tools may provide specialized processing for this string.

The form with a single argument (or two arguments if Reason present), where the first argument is `ON` or `OFF` may be used as a configuration pragma.

If the `LOCAL_NAME` parameter is present, warnings are suppressed for the specified entity. This suppression is effective from the point where it occurs till the end of the extended scope of the variable (similar to the scope of `Suppress`). This form cannot be used as a configuration pragma.

In the case where the first argument is other than `ON` or `OFF`, the third form with a single static_string_EXPRESSION argument (and possible reason) provides more precise control over which warnings are active. The string is a list of letters specifying which warnings are to be activated and which deactivated. The code for these letters is the same as the string used in the command line switch controlling warnings. For a brief summary, use the gnatmake command with no arguments, which will generate usage information containing the list of warnings switches supported. For full details see the section on `Warning Message Control` in the *GNAT User's Guide*. This form can also be used as a configuration pragma.

The warnings controlled by the `-gnatw` switch are generated by the front end of the compiler. The GCC back end can provide additional warnings and they are controlled by the `-W` switch. Such warnings can be identified by the appearance of a string of the form `[-W{xxx}]` in the message which designates the ``-W`xxx'`` switch that controls the message. The form with a single 'static_string_EXPRESSION' argument also works for these warnings, but the string must be a single full ``-W`xxx'`` switch in this case. The above reference lists a few examples of these additional warnings.

The specified warnings will be in effect until the end of the program or another pragma `Warnings` is encountered. The effect of the pragma is cumulative. Initially the set of warnings is the standard default set as possibly modified by compiler switches. Then each pragma Warning modifies this set of warnings as specified. This form of the pragma may also be used as a configuration pragma.

The fourth form, with an `On|Off` parameter and a string, is used to control individual messages, based on their text. The string argument is a pattern that is used to match against the text of individual warning messages (not including the initial "warning: " tag).

The pattern may contain asterisks, which match zero or more characters in the message. For example, you can use `pragma Warnings (Off, "bits of*unused")` to suppress the warning message `warning: 960 bits of "a" unused`. No other regular expression notations are permitted. All characters other than asterisk in these three specific cases are treated as literal characters in the match. The match is case insensitive, for example XYZ matches xyz.

Note that the pattern matches if it occurs anywhere within the warning message string (it is not necessary to put an asterisk at the start and the end of the message, since this is implied).

The above use of patterns to match the message applies only to warning messages generated by the front end. This form of the pragma with a string argument can also be used to control warnings provided by the back end and mentioned above. By using a single full `-W`xxx'` switch in the pragma, such warnings can be turned on and off.

There are two ways to use the pragma in this form. The OFF form can be used as a configuration pragma. The effect is to suppress all warnings (if any) that match the pattern string throughout the compilation (or match the -W switch in the back end case).

The second usage is to suppress a warning locally, and in this case, two pragmas must appear in sequence:

```
pragma Warnings (Off, Pattern);
... code where given warning is to be suppressed
pragma Warnings (On, Pattern);
```

In this usage, the pattern string must match in the Off and On pragmas, and (if '-gnatw.w' is given) at least one matching warning must be suppressed.

Note: if the ON form is not found, then the effect of the OFF form extends until the end of the file (pragma Warnings is purely textual, so its effect does not stop at the end of the enclosing scope).

Note: to write a string that will match any warning, use the string `"***"`. It will not work to use a single asterisk or two asterisks since this looks like an operator name. This form with three asterisks is similar in effect to specifying `pragma Warnings (Off)` except (if `-gnatw.w` is given) that a matching `pragma Warnings (On, "***")` will be required. This can be helpful in avoiding forgetting to turn warnings back on.

Note: the debug flag `-gnatd.i` can be used to cause the compiler to entirely ignore all WARNINGS pragmas. This can be useful in checking whether obsolete pragmas in existing programs are hiding real problems.

Note: pragma Warnings does not affect the processing of style messages. See separate entry for pragma Style_Checks for control of style messages.

Users of the formal verification tool GNATprove for the SPARK subset of Ada may use the version of the pragma with a `TOOL_NAME` parameter.

If present, `TOOL_NAME` is the name of a tool, currently either `GNAT` for the compiler or `GNATprove` for the formal verification tool. A given tool only takes into account pragma Warnings that do not specify a tool name, or that specify the matching tool name. This makes it possible to disable warnings selectively for each tool, and as a consequence to detect useless pragma Warnings with switch `-gnatw.w`.

2.209 Pragma Weak_External

Syntax:

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

`LOCAL_NAME` must refer to an object that is declared at the library level. This pragma specifies that the given entity should be marked as a weak symbol for the linker. It is equivalent to `__attribute__((weak))` in GNU C and causes `LOCAL_NAME` to be emitted as a weak symbol instead of a regular symbol, that is to say a symbol that does not have to be resolved by the linker if used in conjunction with a pragma Import.

When a weak symbol is not resolved by the linker, its address is set to zero. This is useful in writing interfaces to external modules that may or may not be linked in the final executable, for example depending on configuration settings.

If a program references at run time an entity to which this pragma has been applied, and the corresponding symbol was not resolved at link time, then the execution of the program is erroneous. It is not erroneous to take the Address of such an entity, for example to guard potential references, as shown in the example below.

Some file formats do not support weak symbols so not all target machines support this pragma.

```
--  Example of the use of pragma Weak_External

package External_Module is
  key : Integer;
  pragma Import (C, key);
  pragma Weak_External (key);
  function Present return boolean;
end External_Module;

with System; use System;
package body External_Module is
  function Present return boolean is
  begin
    return key'Address /= System.Null_Address;
  end Present;
end External_Module;
```

2.210 Pragma Wide_Character_Encoding

Syntax:

```
pragma Wide_Character_Encoding (IDENTIFIER | CHARACTER_LITERAL);
```

This pragma specifies the wide character encoding to be used in program source text appearing subsequently. It is a configuration pragma, but may also be used at any point that a pragma is allowed, and it is permissible to have more than one such pragma in a file, allowing multiple encodings to appear within the same file.

However, note that the pragma cannot immediately precede the relevant wide character, because then the previous encoding will still be in effect, causing “illegal character” errors.

The argument can be an identifier or a character literal. In the identifier case, it is one of `HEX`, `UPPER`, `SHIFT_JIS`, `EUC`, `UTF8`, or `BRACKETS`. In the character literal case it is correspondingly one of the characters `h`, `u`, `s`, `e`, `8`, or `b`.

Note that when the pragma is used within a file, it affects only the encoding within that file, and does not affect withed units, specs, or subunits.

3 Implementation Defined Aspects

Ada defines (throughout the Ada 2012 reference manual, summarized in Annex K) a set of aspects that can be specified for certain entities. These language defined aspects are implemented in GNAT in Ada 2012 mode and work as described in the Ada 2012 Reference Manual.

In addition, Ada 2012 allows implementations to define additional aspects whose meaning is defined by the implementation. GNAT provides a number of these implementation-defined aspects which can be used to extend and enhance the functionality of the compiler. This section of the GNAT reference manual describes these additional aspects.

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

Note that for many of these aspects, the effect is essentially similar to the use of a pragma or attribute specification with the same name applied to the entity. For example, if we write:

```
type R is range 1 .. 100
  with Value_Size => 10;
```

then the effect is the same as:

```
type R is range 1 .. 100;
for R'Value_Size use 10;
```

and if we write:

```
type R is new Integer
  with Shared => True;
```

then the effect is the same as:

```
type R is new Integer;
pragma Shared (R);
```

In the documentation below, such cases are simply marked as being boolean aspects equivalent to the corresponding pragma or attribute definition clause.

3.1 Aspect Abstract_State

This aspect is equivalent to [pragma Abstract_State], page 5.

3.2 Aspect Always_Terminates

This boolean aspect is equivalent to [pragma Always_Terminates], page 9.

3.3 Aspect Annotate

There are three forms of this aspect (where ID is an identifier, and ARG is a general expression), corresponding to [pragma Annotate], page 10.

'Annotate => ID'

Equivalent to `pragma Annotate (ID, Entity => Name);`

'Annotate => (ID)'
    Equivalent to `pragma Annotate (ID, Entity => Name);`

'Annotate => (ID ,ID {, ARG})'
    Equivalent to `pragma Annotate (ID, ID {, ARG}, Entity => Name);`

3.4 Aspect Async_Readers

This boolean aspect is equivalent to [pragma Async_Readers], page 14.

3.5 Aspect Async_Writers

This boolean aspect is equivalent to [pragma Async_Writers], page 14.

3.6 Aspect Constant_After_Elaboration

This aspect is equivalent to [pragma Constant_After_Elaboration], page 21.

3.7 Aspect Contract_Cases

This aspect is equivalent to [pragma Contract_Cases], page 21, the sequence of clauses being enclosed in parentheses so that syntactically it is an aggregate.

3.8 Aspect Depends

This aspect is equivalent to [pragma Depends], page 26.

3.9 Aspect Default_Initial_Condition

This aspect is equivalent to [pragma Default_Initial_Condition], page 25.

3.10 Aspect Dimension

The `Dimension` aspect is used to specify the dimensions of a given subtype of a dimensioned numeric type. The aspect also specifies a symbol used when doing formatted output of dimensioned quantities. The syntax is:

```
with Dimension =>
  ([Symbol =>] SYMBOL, DIMENSION_VALUE {, DIMENSION_Value})

SYMBOL ::= STRING_LITERAL | CHARACTER_LITERAL

DIMENSION_VALUE ::=
  RATIONAL
| others               => RATIONAL
| DISCRETE_CHOICE_LIST => RATIONAL

RATIONAL ::= [-] NUMERIC_LITERAL [/ NUMERIC_LITERAL]
```

This aspect can only be applied to a subtype whose parent type has a `Dimension_System` aspect. The aspect must specify values for all dimensions of the system. The rational values are the powers of the corresponding dimensions that are used by the compiler to verify that physical (numeric) computations are dimensionally consistent. For example, the computation of a force must result in dimensions (L => 1, M => 1, T => -2). For further examples of the usage of this aspect, see package `System.Dim.Mks`. Note that when the dimensioned type is an integer type, then any dimension value must be an integer literal.

3.11 Aspect Dimension_System

The `Dimension_System` aspect is used to define a system of dimensions that will be used in subsequent subtype declarations with `Dimension` aspects that reference this system. The syntax is:

```
with Dimension_System => (DIMENSION {, DIMENSION});

DIMENSION ::= ([Unit_Name   =>] IDENTIFIER,
               [Unit_Symbol =>] SYMBOL,
               [Dim_Symbol  =>] SYMBOL)

SYMBOL ::= CHARACTER_LITERAL | STRING_LITERAL
```

This aspect is applied to a type, which must be a numeric derived type (typically a floating-point type), that will represent values within the dimension system. Each `DIMENSION` corresponds to one particular dimension. A maximum of 7 dimensions may be specified. `Unit_Name` is the name of the dimension (for example `Meter`). `Unit_Symbol` is the shorthand used for quantities of this dimension (for example `m` for `Meter`). `Dim_Symbol` gives the identification within the dimension system (typically this is a single letter, e.g. `L` standing for length for unit name `Meter`). The `Unit_Symbol` is used in formatted output of dimensioned quantities. The `Dim_Symbol` is used in error messages when numeric operations have inconsistent dimensions.

GNAT provides the standard definition of the International MKS system in the run-time package `System.Dim.Mks`. You can easily define similar packages for cgs units or British units, and define conversion factors between values in different systems. The MKS system is characterized by the following aspect:

```
type Mks_Type is new Long_Long_Float with
  Dimension_System => (
    (Unit_Name => Meter,    Unit_Symbol => 'm',   Dim_Symbol => 'L'),
    (Unit_Name => Kilogram, Unit_Symbol => "kg",  Dim_Symbol => 'M'),
    (Unit_Name => Second,   Unit_Symbol => 's',   Dim_Symbol => 'T'),
    (Unit_Name => Ampere,   Unit_Symbol => 'A',   Dim_Symbol => 'I'),
    (Unit_Name => Kelvin,   Unit_Symbol => 'K',   Dim_Symbol => '@'),
    (Unit_Name => Mole,     Unit_Symbol => "mol", Dim_Symbol => 'N'),
    (Unit_Name => Candela,  Unit_Symbol => "cd",  Dim_Symbol => 'J'));
```

Note that in the above type definition, we use the `at` symbol (`@`) to represent a theta character (avoiding the use of extended Latin-1 characters in this context).

See section 'Performing Dimensionality Analysis in GNAT' in the GNAT Users Guide for detailed examples of use of the dimension system.

3.12 Aspect Disable_Controlled

The aspect `Disable_Controlled` is defined for controlled record types. If active, this aspect causes suppression of all related calls to `Initialize`, `Adjust`, and `Finalize`. The intended use is for conditional compilation, where for example you might want a record to be controlled or not depending on whether some run-time check is enabled or suppressed.

3.13 Aspect Effective_Reads

This aspect is equivalent to [pragma Effective_Reads], page 28.

3.14 Aspect Effective_Writes

This aspect is equivalent to [pragma Effective_Writes], page 28.

3.15 Aspect Exceptional_Cases

This aspect may be specified for procedures and functions with side effects; it can be used to list exceptions that might be propagated by the subprogram with side effects in the context of its precondition, and associate them with a specific postcondition.

For the syntax and semantics of this aspect, see the SPARK 2014 Reference Manual, section 6.1.9.

3.16 Aspect Exit_Cases

This aspect may be specified for procedures and functions with side effects; it can be used to partition the input state into a list of cases and specify, for each case, how the subprogram is allowed to terminate (i.e. return normally or propagate an exception).

For the syntax and semantics of this aspect, see the SPARK 2014 Reference Manual, section 6.1.10.

3.17 Aspect Extended_Access

This nonoverridable boolean-valued type-related representation aspect can be specified as part of a full_type_declaration for a general access type designating an unconstrained array subtype.

The absence of an Extended_Access aspect specification for such a full_type_declaration is equivalent to an explicit "Extended_Access => False" specification. This implies that the aspect is never unspecified for an eligible access type. An access type for which this aspect is True is said to be an extended access type; this includes the case of a type derived from an extended access type. Similarly, a value of such a type is said to be an extended access value.

The representation of an extended access value is different than that of other access values. This representation makes it possible to designate objects that cannot be designated using the usual "thin" or "fat" access representations for an access type designating an unconstrained array subtype (notably slices and array objects imported from other languages).

In particular, two rules are modified in determining the legality of an Access or Unchecked_Access attribute reference if the expected access type is an extended access type:

* A slice of an aliased array object of a non-bitpacked type (more precisely, of an array type having independently addressable components) is considered to be aliased (and the accessibility level of a slice of an array object is defined to be that of the array object); this also applies to renamings of such slices, slices of such renamings, etc.
* The requirement that the nominal subtype of the prefix shall statically match the designated subtype of the access type need not be met.

The Size aspect (and other aspects including Stream_Size, Object_Size, and Alignment) of an extended access type may depend on the properties of the designated type. Further details of this dependence are not documented.

An extended access value is not convertible to a non-extended access type, although conversions in the opposite direction are allowed. We don't want to allow

```
type Big_Ref is access all String with Extended_Access;
type Small_Ref is access all String;
Obj : aliased String := "abcde";
Big_Ptr : Big_Ref := Obj (2 .. 4)'Access; -- OK
Small_Ptr : Small_Ref := Small_Ref (Big_Ptr); -- ERROR: illegal conversion
```

because there is no way to represent the result of such a conversion.

A dereference of an extended access value (or a reference to a renaming thereof) shall not occur in any of the following contexts:

* as an operative constituent of the prefix of an Access or Unchecked_Access attribute reference whose expected type is not extended; or
* as an operative constituent of an actual parameter in a call where the corresponding formal parameter is explicitly aliased.

For the same reasons that explicit conversions from an extended access type to a non-extended access type are forbidden, we also need to disallow getting the same effect via a Extended_Ptr.all'Access reference; this includes the case of passing Extended_Ptr.all as an actual parameter in a call where the corresponding formal parameter is explicitly aliased (because the callee could evaluate Formal_Parameter'Access). This goal is accomplished by adjusting the definition of the term "aliased". A dereference of an extended value occurring in one of these contexts is defined to denote a nonaliased view. This has the desired effect because these contexts require an aliased view. Continuing the preceding example, this rule disallows

```
Sneaky_1 : Small_Ptr := Big_Ptr.all'Access; -- ERROR: illegal 'Access prefix

function Make (Str : aliased in out String) return Small_Ptr
  is (Str'Access); -- OK

Sneaky_2 : Small_Ptr := Make (Str => Big_Ptr.all); -- ERROR: bad parameter
```

for the same reason given above in the case of an explicit type conversion.

3.18 Aspect Extensions_Visible

This aspect is equivalent to [pragma Extensions_Visible], page 35.

3.19 Aspect Favor_Top_Level

This boolean aspect is equivalent to [pragma Favor_Top_Level], page 37.

3.20 Aspect Ghost

This aspect is equivalent to [pragma Ghost], page 38.

3.21 Aspect Ghost_Predicate

This aspect introduces a subtype predicate that can reference ghost entities. The subtype cannot appear as a subtype_mark in a membership test.

For the detailed semantics of this aspect, see the entry for subtype predicates in the SPARK Reference Manual, section 3.2.4.

3.22 Aspect Global

This aspect is equivalent to [pragma Global], page 38.

3.23 Aspect Initial_Condition

This aspect is equivalent to [pragma Initial_Condition], page 44.

3.24 Aspect Initializes

This aspect is equivalent to [pragma Initializes], page 46.

3.25 Aspect Inline_Always

This boolean aspect is equivalent to [pragma Inline_Always], page 46.

3.26 Aspect Invariant

This aspect is equivalent to [pragma Invariant], page 49. It is a synonym for the language defined aspect `Type_Invariant` except that it is separately controllable using pragma `Assertion_Policy`.

3.27 Aspect Invariant'Class

This aspect is equivalent to [pragma Type_Invariant_Class], page 97. It is a synonym for the language defined aspect `Type_Invariant'Class` except that it is separately controllable using pragma `Assertion_Policy`.

3.28 Aspect Iterable

This aspect provides a light-weight mechanism for loops and quantified expressions over container types, without the overhead imposed by the tampering checks of standard Ada 2012 iterators. The value of the aspect is an aggregate with six named components, of which the last three are optional: `First`, `Next`, `Has_Element`, `Element`, `Last`, and `Previous`. When only the first three components are specified, only the `for .. in` form of iteration over cursors is available. When `Element` is specified, both this form and the `for .. of` form of iteration over elements are available. If the last two components are specified, reverse iterations over the container can be specified (analogous to what can be done over predefined containers that support the `Reverse_Iterator` interface). The following is a typical example of use:

```
type List is private with
    Iterable => (First        => First_Cursor,
                 Next         => Advance,
                 Has_Element  => Cursor_Has_Element
                [,Element     => Get_Element]
                [,Last        => Last_Cursor]
                [,Previous    => Retreat]);
```

* The values of `First` and `Last` are primitive operations of the container type that return a `Cursor`, which must be a type declared in the container package or visible from it. For example:

```
function First_Cursor (Cont : Container) return Cursor;
function Last_Cursor  (Cont : Container) return Cursor;
```

* The values of `Next` and `Previous` are primitive operations of the container type that take both a container and a cursor and yield a cursor. For example:

```
function Advance (Cont : Container; Position : Cursor) return Cursor;
function Retreat (Cont : Container; Position : Cursor) return Cursor;
```

* The value of `Has_Element` is a primitive operation of the container type that takes both a container and a cursor and yields a boolean. For example:

```
function Cursor_Has_Element (Cont : Container; Position : Cursor) return Boolean;
```

* The value of `Element` is a primitive operation of the container type that takes both a container and a cursor and yields an `Element_Type`, which must be a type declared in the container package or visible from it. For example:

```
function Get_Element (Cont : Container; Position : Cursor) return Element_Type;
```

This aspect is used in the GNAT-defined formal container packages.

3.29 Aspect Linker_Section

This aspect is equivalent to [pragma Linker_Section], page 52.

3.30 Aspect Local_Restrictions

This aspect may be specified for a subprogram (and for other declarations as described below). It is used to specify that a particular subprogram does not violate one or more local restrictions, nor can it call a subprogram that is not subject to the same requirement. Positional aggregate syntax (with parentheses, not square brackets) may be used to specify more than one local restriction, as in

```
procedure Do_Something
  with Local_Restrictions => (Some_Restriction, Another_Restriction);
```

Parentheses are currently required even in the case of specifying a single local restriction (this requirement may be relaxed in the future). Supported local restrictions currently include (only) No_Heap_Allocations and No_Secondary_Stack. No_Secondary_Stack corresponds to the GNAT-defined (global) restriction of the same name. No_Heap_Allocations corresponds to the conjunction of the Ada-defined restrictions No_Allocators and No_Implicit_Heap_Allocations.

Additional requirements are imposed in order to ensure that restriction violations cannot be achieved via overriding dispatching operations, calling through an access-to-subprogram value, calling a generic formal subprogram, or calling through a subprogram renaming. For a dispatching operation, an overrider must be subject to (at least) the same restrictions as the overridden inherited subprogram; similarly, the actual subprogram corresponding to a generic formal subprogram in an instantiation must be subject to (at least) the same restrictions as the formal subprogram. A call through an access-to-subprogram value is conservatively assumed to violate all local restrictions; tasking-related constructs (notably entry calls) are treated similarly. A renaming-as-body is treated like a subprogram body containing a call to the renamed subprogram.

The Local_Restrictions aspect can be specified for a package specification, in which case the aspect specification also applies to all eligible entities declared with the package. This includes types. Default initialization of an object of a given type is treated like a call to an implicitly-declared initialization subprogram. Such a "call" is subject to the same local restriction checks as any other call. If a type is subject to a local restriction, then any violations of that restriction within the default initialization expressions (if any) of the type are rejected. This may include "calls" to the default initialization subprograms of other types.

Local_Restrictions aspect specifications are additive (for example, in the case of a declaration that occurs within nested packages that each have a Local_Restrictions specification).

3.31 Aspect Lock_Free

This boolean aspect is equivalent to [pragma Lock_Free], page 53.

3.32 Aspect Max_Queue_Length

This aspect is equivalent to [pragma Max_Queue_Length], page 57.

3.33 Aspect No_Caching

This boolean aspect is equivalent to [pragma No_Caching], page 57.

3.34 Aspect No_Elaboration_Code_All

This aspect is equivalent to [pragma No_Elaboration_Code_All], page 58, for a program unit.

3.35 Aspect No_Inline

This boolean aspect is equivalent to [pragma No_Inline], page 58.

3.36 Aspect No_Raise

This boolean aspect is equivalent to [pragma No_Raise], page 58.

3.37 Aspect No_Tagged_Streams

This aspect is equivalent to [pragma No_Tagged_Streams], page 59, with an argument specifying a root tagged type (thus this aspect can only be applied to such a type).

3.38 Aspect No_Task_Parts

Applies to a type. If True, requires that the type and any descendants do not have any task parts. The rules for this aspect are the same as for the language-defined No_Controlled_Parts aspect (see RM-H.4.1), replacing "controlled" with "task".

If No_Task_Parts is True for a type T, then the compiler can optimize away certain tasking-related code that would otherwise be needed for T'Class, because descendants of T might contain tasks.

3.39 Aspect Object_Size

This aspect is equivalent to [attribute Object_Size], page 130.

3.40 Aspect Obsolescent

This aspect is equivalent to [pragma Obsolescent], page 61. Note that the evaluation of this aspect happens at the point of occurrence, it is not delayed until the freeze point.

3.41 Aspect Part_Of

This aspect is equivalent to [pragma Part_Of], page 66.

3.42 Aspect Persistent_BSS

This boolean aspect is equivalent to [pragma Persistent_BSS], page 66.

3.43 Aspect Potentially_Invalid

For the syntax and semantics of this aspect, see the SPARK 2014 Reference Manual, section 13.9.1.

3.44 Aspect Predicate

This aspect is equivalent to [pragma Predicate], page 71. It is thus similar to the language defined aspects `Dynamic_Predicate` and `Static_Predicate` except that whether the resulting predicate is static or dynamic is controlled by the form of the expression. It is also separately controllable using pragma `Assertion_Policy`.

3.45 Aspect Program_Exit

This boolean aspect is equivalent to [pragma Program_Exit], page 76.

3.46 Aspect Pure_Function

This boolean aspect is equivalent to [pragma Pure_Function], page 77.

3.47 Aspect Refined_Depends

This aspect is equivalent to [pragma Refined_Depends], page 78.

3.48 Aspect Refined_Global

This aspect is equivalent to [pragma Refined_Global], page 79.

3.49 Aspect Refined_Post

This aspect is equivalent to [pragma Refined_Post], page 79.

3.50 Aspect Refined_State

This aspect is equivalent to [pragma Refined_State], page 79.

3.51 Aspect Relaxed_Initialization

For the syntax and semantics of this aspect, see the SPARK 2014 Reference Manual, section 6.10.

3.52 Aspect Remote_Access_Type

This aspect is equivalent to [pragma Remote_Access_Type], page 80.

3.53 Aspect Scalar_Storage_Order

This aspect is equivalent to a [attribute Scalar_Storage_Order], page 133.

3.54 Aspect Secondary_Stack_Size

This aspect is equivalent to [pragma Secondary_Stack_Size], page 82.

3.55 Aspect Shared

This boolean aspect is equivalent to [pragma Shared], page 83, and is thus a synonym for aspect `Atomic`.

3.56 Aspect Side_Effects

This aspect is equivalent to [pragma Side_Effects], page 84.

3.57 Aspect Simple_Storage_Pool

This aspect is equivalent to [attribute Simple_Storage_Pool], page 136.

3.58 Aspect Simple_Storage_Pool_Type

This boolean aspect is equivalent to [pragma Simple_Storage_Pool_Type], page 84.

3.59 Aspect SPARK_Mode

This aspect is equivalent to [pragma SPARK_Mode], page 87, and may be specified for either or both of the specification and body of a subprogram or package.

3.60 Aspect Subprogram_Variant

For the syntax and semantics of this aspect, see the SPARK 2014 Reference Manual, section 6.1.8.

3.61 Aspect Suppress_Debug_Info

This boolean aspect is equivalent to [pragma Suppress_Debug_Info], page 93.

3.62 Aspect Suppress_Initialization

This boolean aspect is equivalent to [pragma Suppress_Initialization], page 93.

3.63 Aspect Test_Case

This aspect is equivalent to [pragma Test_Case], page 95.

3.64 Aspect Thread_Local_Storage

This boolean aspect is equivalent to [pragma Thread_Local_Storage], page 96.

3.65 Aspect Universal_Aliasing

This boolean aspect is equivalent to [pragma Universal_Aliasing], page 99.

3.66 Aspect Unmodified

This boolean aspect is equivalent to [pragma Unmodified], page 99.

3.67 Aspect Unreferenced

This boolean aspect is equivalent to [pragma Unreferenced], page 100.

When using the `-gnat2022` switch, this aspect is also supported on formal parameters, which is in particular the only form possible for expression functions.

3.68 Aspect Unreferenced_Objects

This boolean aspect is equivalent to [pragma Unreferenced_Objects], page 100.

3.69 Aspect User_Aspect

This aspect takes an argument that is the name of an aspect defined by a User_Aspect_Definition configuration pragma. A User_Aspect aspect specification is semantically equivalent to replicating the set of aspect specifications associated with the named pragma-defined aspect.

3.70 Aspect Value_Size

This aspect is equivalent to [attribute Value_Size], page 144.

3.71 Aspect Volatile_Full_Access

This boolean aspect is equivalent to [pragma Volatile_Full_Access], page 103.

3.72 Aspect Volatile_Function

This boolean aspect is equivalent to [pragma Volatile_Function], page 104.

3.73 Aspect Warnings

This aspect is equivalent to the two argument form of [pragma Warnings], page 105, where the first argument is `ON` or `OFF` and the second argument is the entity.

4 Implementation Defined Attributes

Ada defines (throughout the Ada reference manual, summarized in Annex K), a set of attributes that provide useful additional functionality in all areas of the language. These language defined attributes are implemented in GNAT and work as described in the Ada Reference Manual.

In addition, Ada allows implementations to define additional attributes whose meaning is defined by the implementation. GNAT provides a number of these implementation-dependent attributes which can be used to extend and enhance the functionality of the compiler. This section of the GNAT reference manual describes these additional attributes. It also describes additional implementation-dependent features of standard language-defined attributes.

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

4.1 Attribute Abort_Signal

`Standard'Abort_Signal` (`Standard` is the only allowed prefix) provides the entity for the special exception used to signal task abort or asynchronous transfer of control. Normally this attribute should only be used in the tasking runtime (it is highly peculiar, and completely outside the normal semantics of Ada, for a user program to intercept the abort exception).

4.2 Attribute Address_Size

`Standard'Address_Size` (`Standard` is the only allowed prefix) is a static constant giving the number of bits in an `Address`. It is the same value as System.Address'Size, but has the advantage of being static, while a direct reference to System.Address'Size is nonstatic because Address is a private type.

4.3 Attribute Asm_Input

The `Asm_Input` attribute denotes a function that takes two parameters. The first is a string, the second is an expression of the type designated by the prefix. The first (string) argument is required to be a static expression, and is the constraint for the parameter, (e.g., what kind of register is required). The second argument is the value to be used as the input argument. The possible values for the constant are the same as those used in the RTL, and are dependent on the configuration file used to built the GCC back end. [Machine Code Insertions], page 287,

4.4 Attribute Asm_Output

The `Asm_Output` attribute denotes a function that takes two parameters. The first is a string, the second is the name of a variable of the type designated by the attribute prefix. The first (string) argument is required to be a static expression and designates the constraint for the parameter (e.g., what kind of register is required). The second argument is the variable to be updated with the result. The possible values for constraint are the same as those used in the RTL, and are dependent on the configuration file used to build the GCC back end. If there are no output operands, then this argument may either be omitted, or explicitly given as `No_Output_Operands`. [Machine Code Insertions], page 287,

4.5 Attribute Atomic_Always_Lock_Free

The prefix of the `Atomic_Always_Lock_Free` attribute is a type. The result indicates whether atomic operations are supported by the target for the given type.

4.6 Attribute Bit

`obj'Bit`, where `obj` is any object, yields the bit offset within the storage unit (byte) that contains the first bit of storage allocated for the object. The value of this attribute is of the type 'universal_integer' and is always a nonnegative number smaller than `System.Storage_Unit`.

For an object that is a variable or a constant allocated in a register, the value is zero. (The use of this attribute does not force the allocation of a variable to memory).

For an object that is a formal parameter, this attribute applies to either the matching actual parameter or to a copy of the matching actual parameter.

For an access object the value is zero. Note that `obj.all'Bit` is subject to an `Access_Check` for the designated object. Similarly for a record component `X.C'Bit` is subject to a discriminant check and `X(I).Bit` and `X(I1..I2)'Bit` are subject to index checks.

This attribute is designed to be compatible with the DEC Ada 83 definition and implementation of the `Bit` attribute.

4.7 Attribute Bit_Position

`R.C'Bit_Position`, where `R` is a record object and `C` is one of the fields of the record type, yields the bit offset within the record contains the first bit of storage allocated for the object. The value of this attribute is of the type 'universal_integer'. The value depends only on the field `C` and is independent of the alignment of the containing record `R`.

4.8 Attribute Code_Address

The `'Address` attribute may be applied to subprograms in Ada 95 and Ada 2005, but the intended effect seems to be to provide an address value which can be used to call the subprogram by means of an address clause as in the following example:

```
procedure K is ...

procedure L;
for L'Address use K'Address;
pragma Import (Ada, L);
```

A call to `L` is then expected to result in a call to `K`. In Ada 83, where there were no access-to-subprogram values, this was a common work-around for getting the effect of an indirect call. GNAT implements the above use of `Address` and the technique illustrated by the example code works correctly.

However, for some purposes, it is useful to have the address of the start of the generated code for the subprogram. On some architectures, this is not necessarily the same as the `Address` value described above. For example, the `Address` value may reference a subprogram descriptor rather than the subprogram itself.

The `'Code_Address` attribute, which can only be applied to subprogram entities, always returns the address of the start of the generated code of the specified subprogram, which may or may not be the same value as is returned by the corresponding `'Address` attribute.

4.9 Attribute Compiler_Version

`Standard'Compiler_Version` (`Standard` is the only allowed prefix) yields a static string identifying the version of the compiler being used to compile the unit containing the attribute reference.

4.10 Attribute Constrained

In addition to the usage of this attribute in the Ada RM, GNAT also permits the use of the `'Constrained` attribute in a generic template for any type, including types without discriminants. The value of this attribute in the generic instance when applied to a scalar type or a record type without discriminants is always `True`. This usage is compatible with older Ada compilers, including notably DEC Ada.

4.11 Attribute Default_Bit_Order

`Standard'Default_Bit_Order` (`Standard` is the only allowed prefix), provides the value `System.Default_Bit_Order` as a `Pos` value (0 for `High_Order_First`, 1 for `Low_Order_First`). This is used to construct the definition of `Default_Bit_Order` in package `System`.

4.12 Attribute Default_Scalar_Storage_Order

`Standard'Default_Scalar_Storage_Order` (`Standard` is the only allowed prefix), provides the current value of the default scalar storage order (as specified using pragma `Default_Scalar_Storage_Order`, or equal to `Default_Bit_Order` if unspecified) as a `System.Bit_Order` value. This is a static attribute.

4.13 Attribute Deref

The attribute `typ'Deref(expr)` where `expr` is of type `System.Address` yields the variable of type `typ` that is located at the given address. It is similar to `(totyp (expr).all)`, where `totyp` is an unchecked conversion from address to a named access-to-*typ* type, except that it yields a variable, so it can be used on the left side of an assignment.

4.14 Attribute Descriptor_Size

Nonstatic attribute `Descriptor_Size` returns the size in bits of the descriptor allocated for a type. The result is non-zero only for unconstrained array types and the returned value is of type universal integer. In GNAT, an array descriptor contains bounds information and is located immediately before the first element of the array.

```
type Unconstr_Array is array (Short_Short_Integer range <>) of Positive;
```

```
Put_Line ("Descriptor size = " & Unconstr_Array'Descriptor_Size'Img);
```

The attribute takes into account any padding due to the alignment of the component type. In the example above, the descriptor contains two values of type `Short_Short_Integer` representing the low and high bound. But, since `Positive` has an alignment of 4, the size of the descriptor is `2 * Short_Short_Integer'Size` rounded up to the next multiple of 32, which yields a size of 32 bits, i.e. including 16 bits of padding.

4.15 Attribute Elaborated

The prefix of the `'Elaborated` attribute must be a unit name. The value is a Boolean which indicates whether or not the given unit has been elaborated. This attribute is primarily intended for internal use by the generated code for dynamic elaboration checking, but it can also be used in user programs. The value will always be True once elaboration of all units has been completed. An exception is for units which need no elaboration, the value is always False for such units.

4.16 Attribute Elab_Body

This attribute can only be applied to a program unit name. It returns the entity for the corresponding elaboration procedure for elaborating the body of the referenced unit. This is used in the main generated elaboration procedure by the binder and is not normally used in any other context. However, there may be specialized situations in which it is useful to be able to call this elaboration procedure from Ada code, e.g., if it is necessary to do selective re-elaboration to fix some error.

4.17 Attribute Elab_Spec

This attribute can only be applied to a program unit name. It returns the entity for the corresponding elaboration procedure for elaborating the spec of the referenced unit. This is used in the main generated elaboration procedure by the binder and is not normally used in any other context. However, there may be specialized situations in which it is useful to be able to call this elaboration procedure from Ada code, e.g., if it is necessary to do selective re-elaboration to fix some error.

4.18 Attribute Elab_Subp_Body

This attribute can only be applied to a library level subprogram name and is only allowed in CodePeer mode. It returns the entity for the corresponding elaboration procedure for elaborating the body of the referenced subprogram unit. This is used in the main generated elaboration procedure by the binder in CodePeer mode only and is unrecognized otherwise.

4.19 Attribute Emax

The `Emax` attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.20 Attribute Enabled

The `Enabled` attribute allows an application program to check at compile time to see if the designated check is currently enabled. The prefix is a simple identifier, referencing any predefined check name (other than `All_Checks`) or a check name introduced by pragma Check_Name. If no argument is given for the attribute, the check is for the general state of the check, if an argument is given, then it is an entity name, and the check indicates whether an `Suppress` or `Unsuppress` has been given naming the entity (if not, then the argument is ignored).

Note that instantiations inherit the check status at the point of the instantiation, so a useful idiom is to have a library package that introduces a check name with `pragma Check_Name`, and then contains generic packages or subprograms which use the `Enabled` attribute to see if the check is enabled. A user of this package can then issue a `pragma Suppress` or `pragma Unsuppress` before instantiating the package or subprogram, controlling whether the check will be present.

4.21 Attribute Enum_Rep

Note that this attribute is now standard in Ada 202x and is available as an implementation defined attribute for earlier Ada versions.

For every enumeration subtype `S`, `S'Enum_Rep` denotes a function with the following spec:

```
function S'Enum_Rep (Arg : S'Base) return <Universal_Integer>;
```

It is also allowable to apply `Enum_Rep` directly to an object of an enumeration type or to a non-overloaded enumeration literal. In this case `S'Enum_Rep` is equivalent to `typ'Enum_Rep(S)` where `typ` is the type of the enumeration literal or object.

The function returns the representation value for the given enumeration value. This will be equal to value of the `Pos` attribute in the absence of an enumeration representation clause. This is a static attribute (i.e., the result is static if the argument is static).

`S'Enum_Rep` can also be used with integer types and objects, in which case it simply returns the integer value. The reason for this is to allow it to be used for `(<>)` discrete formal arguments in a generic unit that can be instantiated with either enumeration types or integer types. Note that if `Enum_Rep` is used on a modular type whose upper bound exceeds the upper bound of the largest signed integer type, and the argument is a variable, so that the universal integer calculation is done at run time, then the call to `Enum_Rep` may raise `Constraint_Error`.

4.22 Attribute Enum_Val

Note that this attribute is now standard in Ada 202x and is available as an implementation defined attribute for earlier Ada versions.

For every enumeration subtype `S`, `S'Enum_Val` denotes a function with the following spec:

```
function S'Enum_Val (Arg : <Universal_Integer>) return S'Base;
```

The function returns the enumeration value whose representation matches the argument, or raises Constraint_Error if no enumeration literal of the type has the matching value. This will be equal to value of the `Val` attribute in the absence of an enumeration representation clause. This is a static attribute (i.e., the result is static if the argument is static).

4.23 Attribute Epsilon

The `Epsilon` attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.24 Attribute Fast_Math

`Standard'Fast_Math` (`Standard` is the only allowed prefix) yields a static Boolean value that is True if pragma `Fast_Math` is active, and False otherwise.

4.25 Attribute Finalization_Size

The prefix of attribute `Finalization_Size` must be an object or a non-class-wide type. This attribute returns the size of any hidden data reserved by the compiler to handle finalization-related actions. The type of the attribute is 'universal_integer'.

`Finalization_Size` yields a value of zero for a type with no controlled parts, an object whose type has no controlled parts, or an object of a class-wide type whose tag denotes a type with no controlled parts.

Note that only heap-allocated objects contain finalization data.

4.26 Attribute Fixed_Value

For every fixed-point type `S`, `S'Fixed_Value` denotes a function with the following specification:

```
function S'Fixed_Value (Arg : <Universal_Integer>) return S;
```

The value returned is the fixed-point value `V` such that:

```
V = Arg * S'Small
```

The effect is thus similar to first converting the argument to the integer type used to represent `S`, and then doing an unchecked conversion to the fixed-point type. The difference is that there are full range checks, to ensure that the result is in range. This attribute is primarily intended for use in implementation of the input-output functions for fixed-point values.

4.27 Attribute From_Address

The prefix of this attribute must be a general access-to-array type (or subtype); the attribute takes a System.Address argument and possibly some additional arguments (described below) and yields a value of the given access type that designates an array object located at the given address. In the case of a non-null array this means that the given address is the address of the first element of the array object (not the address of any sort of bounds descriptor). This allows associating bounds with an address that is, for example, passed in from C code.

If the designated array subtype is unconstrained (which is the usual case), then for each dimension (in order) the attribute takes either one or two additional arguments of the corresponding index type - one if the index subtype is a fixed lower bound subtype, two (low bound first) otherwise. In this case, the access type shall be an extended access type (see the description of the Extended_Access aspect). These additional arguments specify the bounds of the designated array object.

If the designated array subtype is constrained then no additional arguments are provided and the bounds of the designated object are those of the designated subtype.

Roughly speaking, My_Access_Type'From_Address (Addr, Lo, Hi) is equivalent to a declare expression:

```
(declare
   Obj : aliased Designated_Array_Type (Lo .. Hi)
     with Import, Address => Addr;
begin
   My_Access_Type'(Obj'Unchecked_Access)
end)
```

4.28 Attribute From_Any

This internal attribute is used for the generation of remote subprogram stubs in the context of the Distributed Systems Annex.

4.29 Attribute Has_Access_Values

The prefix of the `Has_Access_Values` attribute is a type. The result is a Boolean value which is True if the is an access type, or is a composite type with a component (at any nesting depth) that is an access type, and is False otherwise. The intended use of this attribute is in conjunction with generic definitions. If the attribute is applied to a generic private type, it indicates whether or not the corresponding actual type has access values.

4.30 Attribute Has_Discriminants

The prefix of the `Has_Discriminants` attribute is a type. The result is a Boolean value which is True if the type has discriminants, and False otherwise. The intended use of this attribute is in conjunction with generic definitions. If the attribute is applied to a generic private type, it indicates whether or not the corresponding actual type has discriminants.

4.31 Attribute Has_Tagged_Values

The prefix of the `Has_Tagged_Values` attribute is a type. The result is a Boolean value which is True if the type is a composite type (array or record) that is either a tagged type or has a subcomponent that is tagged, and is False otherwise. The intended use of this attribute is in conjunction with generic definitions. If the attribute is applied to a generic private type, it indicates whether or not the corresponding actual type has access values.

4.32 Attribute Img

The `Img` attribute differs from `Image` in that, while both can be applied directly to an object, `Img` cannot be applied to types.

Example usage of the attribute:

```
Put_Line ("X = " & X'Img);
```

which has the same meaning as the more verbose:

```
Put_Line ("X = " & T'Image (X));
```

where `T` is the (sub)type of the object `X`.

Note that technically, in analogy to `Image`, `X'Img` returns a parameterless function that returns the appropriate string when called. This means that `X'Img` can be renamed as a function-returning-string, or used in an instantiation as a function parameter.

4.33 Attribute Initialized

For the syntax and semantics of this attribute, see the SPARK 2014 Reference Manual, section 6.10.

4.34 Attribute Integer_Value

For every integer type `S`, `S'Integer_Value` denotes a function with the following spec:

```
function S'Integer_Value (Arg : <Universal_Fixed>) return S;
```

The value returned is the integer value `V`, such that:

```
Arg = V * T'Small
```

where `T` is the type of `Arg`. The effect is thus similar to first doing an unchecked conversion from the fixed-point type to its corresponding implementation type, and then converting the result to the target integer type. The difference is that there are full range checks, to ensure that the result is in range. This attribute is primarily intended for use in implementation of the standard input-output functions for fixed-point values.

4.35 Attribute Invalid_Value

For every scalar type S, S'Invalid_Value returns an undefined value of the type. If possible this value is an invalid representation for the type. The value returned is identical to the value used to initialize an otherwise uninitialized value of the type if pragma Initialize_Scalars is used, including the ability to modify the value with the binder -Sxx flag and relevant environment variables at run time.

4.36 Attribute Large

The `Large` attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.37 Attribute Library_Level

`P'Library_Level`, where P is an entity name, returns a Boolean value which is True if the entity is declared at the library level, and False otherwise. Note that within a generic instantiation, the name of the generic unit denotes the instance, which means that this attribute can be used to test if a generic is instantiated at the library level, as shown in this example:

```
generic
  ...
package Gen is
  pragma Compile_Time_Error
    (not Gen'Library_Level,
```

```
           "Gen can only be instantiated at library level");
         ...
       end Gen;
```

4.38 Attribute Loop_Entry

Syntax:

```
X'Loop_Entry [(loop_name)]
```

The Loop_Entry attribute is used to refer to the value that an expression had upon entry to a given loop in much the same way that the Old attribute in a subprogram postcondition can be used to refer to the value an expression had upon entry to the subprogram. The relevant loop is either identified by the given loop name, or it is the innermost enclosing loop when no loop name is given.

A Loop_Entry attribute can only occur within an Assert, Assert_And_Cut, Assume, Loop_Variant or Loop_Invariant pragma. In addition, such a pragma must be one of the items in the sequence of statements of a loop body, or nested inside block statements that appear in the sequence of statements of a loop body. A common use of Loop_Entry is to compare the current value of objects with their initial value at loop entry, in a Loop_Invariant pragma.

The effect of using X'Loop_Entry is the same as declaring a constant initialized with the initial value of X at loop entry. This copy is not performed if the loop is not entered, or if the corresponding pragmas are ignored or disabled.

4.39 Attribute Machine_Size

This attribute is identical to the Object_Size attribute. It is provided for compatibility with the DEC Ada 83 attribute of this name.

4.40 Attribute Mantissa

The Mantissa attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.41 Attribute Maximum_Alignment

Standard'Maximum_Alignment (Standard is the only allowed prefix) provides the maximum default alignment value for the target, that is to say the maximum alignment that the compiler may choose by default for a type or an object. Larger alignments are supported up to some maximum value dependent on the target, but may require specific mechanisms that are not needed up to Standard'Maximum_Alignment.

4.42 Attribute Max_Integer_Size

Standard'Max_Integer_Size (Standard is the only allowed prefix) provides the size of the largest supported integer type for the target. The result is a static constant.

4.43 Attribute Mechanism_Code

`func'Mechanism_Code` yields an integer code for the mechanism used for the result of function `func`, and `subprog'Mechanism_Code (n)` yields the mechanism used for formal parameter number 'n' (a static integer value, with 1 meaning the first parameter) of subprogram `subprog`. The code returned is:

'1'

by copy (value)

'2'

by reference

4.44 Attribute Null_Parameter

A reference `T'Null_Parameter` denotes an imaginary object of type or subtype `T` allocated at machine address zero. The attribute is allowed only as the default expression of a formal parameter, or as an actual expression of a subprogram call. In either case, the subprogram must be imported.

The identity of the object is represented by the address zero in the argument list, independent of the passing mechanism (explicit or default).

This capability is needed to specify that a zero address should be passed for a record or other composite object passed by reference. There is no way of indicating this without the `Null_Parameter` attribute.

4.45 Attribute Object_Size

The size of an object is not necessarily the same as the size of the type of an object. This is because by default object sizes are increased to be a multiple of the alignment of the object. For example, `Natural'Size` is 31, but by default objects of type `Natural` will have a size of 32 bits. Similarly, a record containing an integer and a character:

```
type Rec is record
   I : Integer;
   C : Character;
end record;
```

will have a size of 40 (that is `Rec'Size` will be 40). The alignment will be 4, because of the integer field, and so the default size of record objects for this type will be 64 (8 bytes).

If the alignment of the above record is specified to be 1, then the object size will be 40 (5 bytes). This is true by default, and also an object size of 40 can be explicitly specified in this case.

A consequence of this capability is that different object sizes can be given to subtypes that would otherwise be considered in Ada to be statically matching. But it makes no sense to consider such subtypes as statically matching. Consequently, GNAT adds a rule to the static matching rules that requires object sizes to match. Consider this example:

```
 1. procedure BadAVConvert is
 2.    type R is new Integer;
 3.    subtype R1 is R range 1 .. 10;
```

```
 4.    subtype R2 is R range 1 .. 10;
 5.    for R1'Object_Size use 8;
 6.    for R2'Object_Size use 16;
 7.    type R1P is access all R1;
 8.    type R2P is access all R2;
 9.    R1PV : R1P := new R1'(4);
10.    R2PV : R2P;
11. begin
12.    R2PV := R2P (R1PV);
                  |
       >>> target designated subtype not compatible with
           type "R1" defined at line 3

13. end;
```

In the absence of lines 5 and 6, types `R1` and `R2` statically match and hence the conversion on line 12 is legal. But since lines 5 and 6 cause the object sizes to differ, GNAT considers that types `R1` and `R2` are not statically matching, and line 12 generates the diagnostic shown above.

Similar additional checks are performed in other contexts requiring statically matching subtypes.

4.46 Attribute Old

In addition to the usage of `Old` defined in the Ada 2012 RM (usage within `Post` aspect), GNAT also permits the use of this attribute in implementation defined pragmas `Postcondition`, `Contract_Cases` and `Test_Case`. Also usages of `Old` which would be illegal according to the Ada 2012 RM definition are allowed under control of implementation defined pragma `Unevaluated_Use_Of_Old`.

4.47 Attribute Passed_By_Reference

`typ'Passed_By_Reference` for any subtype *typ* returns a value of type `Boolean` value that is `True` if the type is normally passed by reference and `False` if the type is normally passed by copy in calls. For scalar types, the result is always `False` and is static. For non-scalar types, the result is nonstatic.

4.48 Attribute Pool_Address

`X'Pool_Address` for any object `X` returns the address of X within its storage pool. This is the same as `X'Address`, except that for an unconstrained array whose bounds are allocated just before the first component, `X'Pool_Address` returns the address of those bounds, whereas `X'Address` returns the address of the first component.

Here, we are interpreting 'storage pool' broadly to mean `wherever the object is allocated`, which could be a user-defined storage pool, the global heap, on the stack, or in a static memory area. For an object created by `new`, `Ptr.all'Pool_Address` is what is passed to `Allocate` and returned from `Deallocate`.

4.49 Attribute Range_Length

`typ'Range_Length` for any discrete type *typ* yields the number of values represented by the subtype (zero for a null range). The result is static for static subtypes. `Range_Length` applied to the index subtype of a one dimensional array always gives the same result as `Length` applied to the array itself.

4.50 Attribute Restriction_Set

This attribute allows compile time testing of restrictions that are currently in effect. It is primarily intended for specializing code in the run-time based on restrictions that are active (e.g. don't need to save fpt registers if restriction No_Floating_Point is known to be in effect), but can be used anywhere.

There are two forms:

```
System'Restriction_Set (partition_boolean_restriction_NAME)
System'Restriction_Set (No_Dependence => library_unit_NAME);
```

In the case of the first form, the only restriction names allowed are parameterless restrictions that are checked for consistency at bind time. For a complete list see the subtype `System.Rident.Partition_Boolean_Restrictions`.

The result returned is True if the restriction is known to be in effect, and False if the restriction is known not to be in effect. An important guarantee is that the value of a Restriction_Set attribute is known to be consistent throughout all the code of a partition.

This is trivially achieved if the entire partition is compiled with a consistent set of restriction pragmas. However, the compilation model does not require this. It is possible to compile one set of units with one set of pragmas, and another set of units with another set of pragmas. It is even possible to compile a spec with one set of pragmas, and then WITH the same spec with a different set of pragmas. Inconsistencies in the actual use of the restriction are checked at bind time.

In order to achieve the guarantee of consistency for the Restriction_Set pragma, we consider that a use of the pragma that yields False is equivalent to a violation of the restriction.

So for example if you write

```
if System'Restriction_Set (No_Floating_Point) then
   ...
else
   ...
end if;
```

And the result is False, so that the else branch is executed, you can assume that this restriction is not set for any unit in the partition. This is checked by considering this use of the restriction pragma to be a violation of the restriction No_Floating_Point. This means that no other unit can attempt to set this restriction (if some unit does attempt to set it, the binder will refuse to bind the partition).

Technical note: The restriction name and the unit name are intepreted entirely syntactically, as in the corresponding Restrictions pragma, they are not analyzed semantically, so they do not have a type.

4.51 Attribute Result

`function'Result` can only be used with in a Postcondition pragma for a function. The prefix must be the name of the corresponding function. This is used to refer to the result of the function in the postcondition expression. For a further discussion of the use of this attribute and examples of its use, see the description of pragma Postcondition.

4.52 Attribute Round

In addition to the usage of this attribute in the Ada RM, GNAT also permits the use of the `'Round` attribute for ordinary fixed point types.

4.53 Attribute Safe_Emax

The `Safe_Emax` attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.54 Attribute Safe_Large

The `Safe_Large` attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.55 Attribute Safe_Small

The `Safe_Small` attribute is provided for compatibility with Ada 83. See the Ada 83 reference manual for an exact description of the semantics of this attribute.

4.56 Attribute Scalar_Storage_Order

For every array or record type `S`, the representation attribute `Scalar_Storage_Order` denotes the order in which storage elements that make up scalar components are ordered within S. The value given must be a static expression of type System.Bit_Order. The following is an example of the use of this feature:

```
--  Component type definitions

subtype Yr_Type is Natural range 0 .. 127;
subtype Mo_Type is Natural range 1 .. 12;
subtype Da_Type is Natural range 1 .. 31;

--  Record declaration

type Date is record
   Years_Since_1980 : Yr_Type;
   Month            : Mo_Type;
   Day_Of_Month     : Da_Type;
end record;

--  Record representation clause
```

```
for Date use record
   Years_Since_1980 at 0 range 0  ..  6;
   Month            at 0 range 7  .. 10;
   Day_Of_Month     at 0 range 11 .. 15;
end record;

--  Attribute definition clauses

for Date'Bit_Order use System.High_Order_First;
for Date'Scalar_Storage_Order use System.High_Order_First;
--  If Scalar_Storage_Order is specified, it must be consistent with
--  Bit_Order, so it's best to always define the latter explicitly if
--  the former is used.
```

Other properties are as for the standard representation attribute `Bit_Order` defined by Ada RM 13.5.3(4). The default is `System.Default_Bit_Order`.

For a record type `T`, if `T'Scalar_Storage_Order` is specified explicitly, it shall be equal to `T'Bit_Order`. Note: this means that if a `Scalar_Storage_Order` attribute definition clause is not confirming, then the type's `Bit_Order` shall be specified explicitly and set to the same value.

Derived types inherit an explicitly set scalar storage order from their parent types. This may be overridden for the derived type by giving an explicit scalar storage order for it. However, for a record extension, the derived type must have the same scalar storage order as the parent type.

A component of a record type that is itself a record or an array and that does not start and end on a byte boundary must have have the same scalar storage order as the record type. A component of a bit-packed array type that is itself a record or an array must have the same scalar storage order as the array type.

No component of a type that has an explicit `Scalar_Storage_Order` attribute definition may be aliased.

A confirming `Scalar_Storage_Order` attribute definition clause (i.e. with a value equal to `System.Default_Bit_Order`) has no effect.

If the opposite storage order is specified, then whenever the value of a scalar component of an object of type `S` is read, the storage elements of the enclosing machine scalar are first reversed (before retrieving the component value, possibly applying some shift and mask operatings on the enclosing machine scalar), and the opposite operation is done for writes.

In that case, the restrictions set forth in 13.5.1(10.3/2) for scalar components are relaxed. Instead, the following rules apply:

* the underlying storage elements are those at positions `(position + first_bit / storage_element_size) .. (position + (last_bit + storage_element_size - 1) / storage_element_size)`
* the sequence of underlying storage elements shall have a size no greater than the largest machine scalar
* the enclosing machine scalar is defined as the smallest machine scalar starting at a position no greater than `position + first_bit / storage_element_size` and covering

storage elements at least up to `position + (last_bit + storage_element_size - 1) / storage_element_size`

* the position of the component is interpreted relative to that machine scalar.

If no scalar storage order is specified for a type (either directly, or by inheritance in the case of a derived type), then the default is normally the native ordering of the target, but this default can be overridden using pragma `Default_Scalar_Storage_Order`.

If a component of `T` is itself of a record or array type, the specfied `Scalar_Storage_Order` does 'not' apply to that nested type: an explicit attribute definition clause must be provided for the component type as well if desired.

Representation changes that explicitly or implicitly toggle the scalar storage order are not supported and may result in erroneous execution of the program, except when performed by means of an instance of `Ada.Unchecked_Conversion`.

In particular, overlays are not supported and a warning is given for them:

```
type Rec_LE is record
   I : Integer;
end record;

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

for Rec_LE'Bit_Order use System.Low_Order_First;
for Rec_LE'Scalar_Storage_Order use System.Low_Order_First;

type Rec_BE is record
   I : Integer;
end record;

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

for Rec_BE'Bit_Order use System.High_Order_First;
for Rec_BE'Scalar_Storage_Order use System.High_Order_First;

R_LE : Rec_LE;

R_BE : Rec_BE;
for R_BE'Address use R_LE'Address;
```

`warning: overlay changes scalar storage order [enabled by default]`

In most cases, such representation changes ought to be replaced by an instantiation of a function or procedure provided by `GNAT.Byte_Swapping`.

Note that the scalar storage order only affects the in-memory data representation. It has no effect on the representation used by stream attributes.

Note that debuggers may be unable to display the correct value of scalar components of a type for which the opposite storage order is specified.

4.57 Attribute Simple_Storage_Pool

For every nonformal, nonderived access-to-object type `Acc`, the representation attribute `Simple_Storage_Pool` may be specified via an attribute_definition_clause (or by specifying the equivalent aspect):

```
My_Pool : My_Simple_Storage_Pool_Type;

type Acc is access My_Data_Type;

for Acc'Simple_Storage_Pool use My_Pool;
```

The name given in an attribute_definition_clause for the `Simple_Storage_Pool` attribute shall denote a variable of a 'simple storage pool type' (see pragma *Simple_Storage_Pool_Type*).

The use of this attribute is only allowed for a prefix denoting a type for which it has been specified. The type of the attribute is the type of the variable specified as the simple storage pool of the access type, and the attribute denotes that variable.

It is illegal to specify both `Storage_Pool` and `Simple_Storage_Pool` for the same access type.

If the `Simple_Storage_Pool` attribute has been specified for an access type, then applying the `Storage_Pool` attribute to the type is flagged with a warning and its evaluation raises the exception `Program_Error`.

If the Simple_Storage_Pool attribute has been specified for an access type `S`, then the evaluation of the attribute `S'Storage_Size` returns the result of calling `Storage_Size (S'Simple_Storage_Pool)`, which is intended to indicate the number of storage elements reserved for the simple storage pool. If the Storage_Size function has not been defined for the simple storage pool type, then this attribute returns zero.

If an access type `S` has a specified simple storage pool of type `SSP`, then the evaluation of an allocator for that access type calls the primitive `Allocate` procedure for type `SSP`, passing `S'Simple_Storage_Pool` as the pool parameter. The detailed semantics of such allocators is the same as those defined for allocators in section 13.11 of the *Ada Reference Manual*, with the term 'simple storage pool' substituted for 'storage pool'.

If an access type `S` has a specified simple storage pool of type `SSP`, then a call to an instance of the `Ada.Unchecked_Deallocation` for that access type invokes the primitive `Deallocate` procedure for type `SSP`, passing `S'Simple_Storage_Pool` as the pool parameter. The detailed semantics of such unchecked deallocations is the same as defined in section 13.11.2 of the Ada Reference Manual, except that the term 'simple storage pool' is substituted for 'storage pool'.

4.58 Attribute Small

The `Small` attribute is defined in Ada 95 (and Ada 2005) only for fixed-point types. GNAT also allows this attribute to be applied to floating-point types for compatibility with Ada 83.

See the Ada 83 reference manual for an exact description of the semantics of this attribute when applied to floating-point types.

4.59 Attribute Small_Denominator

`typ'Small_Denominator` for any fixed-point subtype *typ* yields the denominator in the representation of `typ'Small` as a rational number with coprime factors (i.e. as an irreducible fraction).

4.60 Attribute Small_Numerator

`typ'Small_Numerator` for any fixed-point subtype *typ* yields the numerator in the representation of `typ'Small` as a rational number with coprime factors (i.e. as an irreducible fraction).

4.61 Attribute Storage_Unit

`Standard'Storage_Unit` (`Standard` is the only allowed prefix) provides the same value as `System.Storage_Unit`.

4.62 Attribute Stub_Type

The GNAT implementation of remote access-to-classwide types is organized as described in AARM section E.4 (20.t): a value of an RACW type (designating a remote object) is represented as a normal access value, pointing to a "stub" object which in turn contains the necessary information to contact the designated remote object. A call on any dispatching operation of such a stub object does the remote call, if necessary, using the information in the stub object to locate the target partition, etc.

For a prefix `T` that denotes a remote access-to-classwide type, `T'Stub_Type` denotes the type of the corresponding stub objects.

By construction, the layout of `T'Stub_Type` is identical to that of type `RACW_Stub_Type` declared in the internal implementation-defined unit `System.Partition_Interface`. Use of this attribute will create an implicit dependency on this unit.

4.63 Attribute System_Allocator_Alignment

`Standard'System_Allocator_Alignment` (`Standard` is the only allowed prefix) provides the observable guaranteed to be honored by the system allocator (malloc). This is a static value that can be used in user storage pools based on malloc either to reject allocation with alignment too large or to enable a realignment circuitry if the alignment request is larger than this value.

4.64 Attribute Target_Name

`Standard'Target_Name` (`Standard` is the only allowed prefix) provides a static string value that identifies the target for the current compilation. For GCC implementations, this is the standard gcc target name without the terminating slash (for example, GNAT 5.0 on windows yields "i586-pc-mingw32msv").

4.65 Attribute To_Address

The `System'To_Address` (`System` is the only allowed prefix) denotes a function identical to `System.Storage_Elements.To_Address` except that it is a static attribute. This means that if its argument is a static expression, then the result of the attribute is a static expression. This means that such an expression can be used in contexts (e.g., preelaborable packages) which require a static expression and where the function call could not be used (since the function call is always nonstatic, even if its argument is static). The argument must be in the range -(2**(m-1)) .. 2**m-1, where m is the memory size (typically 32 or 64). Negative values are intepreted in a modular manner (e.g., -1 means the same as 16#FFFF_FFFF# on a 32 bits machine).

4.66 Attribute To_Any

This internal attribute is used for the generation of remote subprogram stubs in the context of the Distributed Systems Annex.

4.67 Attribute Type_Class

`typ'Type_Class` for any type or subtype *typ* yields the value of the type class for the full type of *typ*. If *typ* is a generic formal type, the value is the value for the corresponding actual subtype. The value of this attribute is of type `System.Aux_DEC.Type_Class`, which has the following definition:

```
type Type_Class is
  (Type_Class_Enumeration,
   Type_Class_Integer,
   Type_Class_Fixed_Point,
   Type_Class_Floating_Point,
   Type_Class_Array,
   Type_Class_Record,
   Type_Class_Access,
   Type_Class_Task,
   Type_Class_Address);
```

Protected types yield the value `Type_Class_Task`, which thus applies to all concurrent types. This attribute is designed to be compatible with the DEC Ada 83 attribute of the same name.

4.68 Attribute Type_Key

The `Type_Key` attribute is applicable to a type or subtype and yields a value of type Standard.String containing encoded information about the type or subtype. This provides improved compatibility with other implementations that support this attribute.

4.69 Attribute TypeCode

This internal attribute is used for the generation of remote subprogram stubs in the context of the Distributed Systems Annex.

4.70 Attribute Unconstrained_Array

The `Unconstrained_Array` attribute can be used with a prefix that denotes any type or subtype. It is a static attribute that yields `True` if the prefix designates an unconstrained array, and `False` otherwise. In a generic instance, the result is still static, and yields the result of applying this test to the generic actual.

4.71 Attribute Universal_Literal_String

The prefix of `Universal_Literal_String` must be a named number. The static result is the string consisting of the characters of the number as defined in the original source. This allows the user program to access the actual text of named numbers without intermediate conversions and without the need to enclose the strings in quotes (which would preclude their use as numbers).

For example, the following program prints the first 50 digits of pi:

```
with Text_IO; use Text_IO;
with Ada.Numerics;
procedure Pi is
begin
   Put (Ada.Numerics.Pi'Universal_Literal_String);
end;
```

4.72 Attribute Unrestricted_Access

The `Unrestricted_Access` attribute is similar to `Access` except that all accessibility and aliased view checks are omitted. This is a user-beware attribute.

For objects, it is similar to `Address`, for which it is a desirable replacement where the value desired is an access type. In other words, its effect is similar to first applying the `Address` attribute and then doing an unchecked conversion to a desired access type.

For subprograms, `P'Unrestricted_Access` may be used where `P'Access` would be illegal, to construct a value of a less-nested named access type that designates a more-nested subprogram. This value may be used in indirect calls, so long as the more-nested subprogram still exists; once the subprogram containing it has returned, such calls are erroneous. For example:

```
package body P is

   type Less_Nested is access procedure;
   Global : Less_Nested;

   procedure P1 is
   begin
      Global.all;
   end P1;

   procedure P2 is
      Local_Var : Integer;
```

```
          procedure More_Nested is
          begin
             ... Local_Var ...
          end More_Nested;
       begin
          Global := More_Nested'Unrestricted_Access;
          P1;
       end P2;

    end P;
```

When P1 is called from P2, the call via Global is OK, but if P1 were called after P2 returns, it would be an erroneous use of a dangling pointer.

For objects, it is possible to use `Unrestricted_Access` for any type. However, if the result is of an access-to-unconstrained array subtype, then the resulting pointer has the same scope as the context of the attribute, and must not be returned to some enclosing scope. For instance, if a function uses `Unrestricted_Access` to create an access-to-unconstrained-array and returns that value to the caller, the result will involve dangling pointers. In addition, it is only valid to create pointers to unconstrained arrays using this attribute if the pointer has the normal default 'fat' representation where a pointer has two components, one points to the array and one points to the bounds. If a size clause is used to force 'thin' representation for a pointer to unconstrained where there is only space for a single pointer, then the resulting pointer is not usable.

In the simple case where a direct use of Unrestricted_Access attempts to make a thin pointer for a non-aliased object, the compiler will reject the use as illegal, as shown in the following example:

```
with System; use System;
procedure SliceUA2 is
   type A is access all String;
   for A'Size use Standard'Address_Size;

   procedure P (Arg : A) is
   begin
      null;
   end P;

   X : String := "hello world!";
   X2 : aliased String := "hello world!";

   AV : A := X'Unrestricted_Access;    -- ERROR
             |
>>> illegal use of Unrestricted_Access attribute
>>> attempt to generate thin pointer to unaliased object

begin
   P (X'Unrestricted_Access);          -- ERROR
      |
```

```
>>> illegal use of Unrestricted_Access attribute
>>> attempt to generate thin pointer to unaliased object

   P (X(7 .. 12)'Unrestricted_Access); -- ERROR
      |
>>> illegal use of Unrestricted_Access attribute
>>> attempt to generate thin pointer to unaliased object

   P (X2'Unrestricted_Access);         -- OK
end;
```

but other cases cannot be detected by the compiler, and are considered to be erroneous. Consider the following example:

```
with System; use System;
with System; use System;
procedure SliceUA is
   type AF is access all String;

   type A is access all String;
   for A'Size use Standard'Address_Size;

   procedure P (Arg : A) is
   begin
      if Arg'Length /= 6 then
         raise Program_Error;
      end if;
   end P;

   X : String := "hello world!";
   Y : AF := X (7 .. 12)'Unrestricted_Access;

begin
   P (A (Y));
end;
```

A normal unconstrained array value or a constrained array object marked as aliased has the bounds in memory just before the array, so a thin pointer can retrieve both the data and the bounds. But in this case, the non-aliased object `X` does not have the bounds before the string. If the size clause for type `A` were not present, then the pointer would be a fat pointer, where one component is a pointer to the bounds, and all would be well. But with the size clause present, the conversion from fat pointer to thin pointer in the call loses the bounds, and so this is erroneous, and the program likely raises a `Program_Error` exception.

In general, it is advisable to completely avoid mixing the use of thin pointers and the use of `Unrestricted_Access` where the designated type is an unconstrained array. The use of thin pointers should be restricted to cases of porting legacy code that implicitly assumes the size of pointers, and such code should not in any case be using this attribute.

Another erroneous situation arises if the attribute is applied to a constant. The resulting pointer can be used to access the constant, but the effect of trying to modify a constant in this manner is not well-defined. Consider this example:

```
P : constant Integer := 4;
type R is access all Integer;
RV : R := P'Unrestricted_Access;
..
RV.all := 3;
```

Here we attempt to modify the constant P from 4 to 3, but the compiler may or may not notice this attempt, and subsequent references to P may yield either the value 3 or the value 4 or the assignment may blow up if the compiler decides to put P in read-only memory. One particular case where `Unrestricted_Access` can be used in this way is to modify the value of an `in` parameter:

```
procedure K (S : in String) is
   type R is access all Character;
   RV : R := S (3)'Unrestricted_Access;
begin
   RV.all := 'a';
end;
```

In general this is a risky approach. It may appear to "work" but such uses of `Unrestricted_Access` are potentially non-portable, even from one version of GNAT to another, so are best avoided if possible.

4.73 Attribute Update

The `Update` attribute creates a copy of an array or record value with one or more modified components. The syntax is:

```
PREFIX'Update ( RECORD_COMPONENT_ASSOCIATION_LIST )
PREFIX'Update ( ARRAY_COMPONENT_ASSOCIATION {, ARRAY_COMPONENT_ASSOCIATION } )
PREFIX'Update ( MULTIDIMENSIONAL_ARRAY_COMPONENT_ASSOCIATION
                {, MULTIDIMENSIONAL_ARRAY_COMPONENT_ASSOCIATION } )

MULTIDIMENSIONAL_ARRAY_COMPONENT_ASSOCIATION ::= INDEX_EXPRESSION_LIST_LIST => EXPRESSION
INDEX_EXPRESSION_LIST_LIST                   ::= INDEX_EXPRESSION_LIST {| INDEX_EXPRESSION_LIST }
INDEX_EXPRESSION_LIST                        ::= ( EXPRESSION {, EXPRESSION } )
```

where `PREFIX` is the name of an array or record object, the association list in parentheses does not contain an `others` choice and the box symbol `<>` may not appear in any expression. The effect is to yield a copy of the array or record value which is unchanged apart from the components mentioned in the association list, which are changed to the indicated value. The original value of the array or record value is not affected. For example:

```
type Arr is Array (1 .. 5) of Integer;
...
Avar1 : Arr := (1,2,3,4,5);
Avar2 : Arr := Avar1'Update (2 => 10, 3 .. 4 => 20);
```

yields a value for `Avar2` of 1,10,20,20,5 with `Avar1` begin unmodified. Similarly:

```
type Rec is A, B, C : Integer;
```

```
...
Rvar1 : Rec := (A => 1, B => 2, C => 3);
Rvar2 : Rec := Rvar1'Update (B => 20);
```

yields a value for `Rvar2` of (A => 1, B => 20, C => 3), with `Rvar1` being unmodifed. Note that the value of the attribute reference is computed completely before it is used. This means that if you write:

```
Avar1 := Avar1'Update (1 => 10, 2 => Function_Call);
```

then the value of `Avar1` is not modified if `Function_Call` raises an exception, unlike the effect of a series of direct assignments to elements of `Avar1`. In general this requires that two extra complete copies of the object are required, which should be kept in mind when considering efficiency.

The `Update` attribute cannot be applied to prefixes of a limited type, and cannot reference discriminants in the case of a record type. The accessibility level of an Update attribute result object is defined as for an aggregate.

In the record case, no component can be mentioned more than once. In the array case, two overlapping ranges can appear in the association list, in which case the modifications are processed left to right.

Multi-dimensional arrays can be modified, as shown by this example:

```
A : array (1 .. 10, 1 .. 10) of Integer;
..
A := A'Update ((1, 2) => 20, (3, 4) => 30);
```

which changes element (1,2) to 20 and (3,4) to 30.

4.74 Attribute Valid_Value

The `'Valid_Value` attribute is defined for enumeration types other than those in package Standard or types derived from those types. This attribute is a function that takes a String, and returns Boolean. `T'Valid_Value (S)` returns True if and only if `T'Value (S)` would not raise Constraint_Error.

4.75 Attribute Valid_Scalars

The `'Valid_Scalars` attribute is intended to make it easier to check the validity of scalar subcomponents of composite objects. The attribute is defined for any prefix `P` which denotes an object. Prefix `P` can be any type except for tagged private or `Unchecked_Union` types. The value of the attribute is of type `Boolean`.

`P'Valid_Scalars` yields `True` if and only if the evaluation of `C'Valid` yields `True` for every scalar subcomponent `C` of `P`, or if `P` has no scalar subcomponents. Attribute `'Valid_Scalars` is equivalent to attribute `'Valid` for scalar types.

It is not specified in what order the subcomponents are checked, nor whether any more are checked after any one of them is determined to be invalid. If the prefix `P` is of a class-wide type `T'Class` (where `T` is the associated specific type), or if the prefix `P` is of a specific tagged type `T`, then only the subcomponents of `T` are checked; in other words, components of extensions of `T` are not checked even if `T'Class (P)'Tag /= T'Tag`.

The compiler will issue a warning if it can be determined at compile time that the prefix of the attribute has no scalar subcomponents.

Note: `Valid_Scalars` can generate a lot of code, especially in the case of a large variant record. If the attribute is called in many places in the same program applied to objects of the same type, it can reduce program size to write a function with a single use of the attribute, and then call that function from multiple places.

4.76 Attribute VADS_Size

The `'VADS_Size` attribute is intended to make it easier to port legacy code which relies on the semantics of `'Size` as implemented by the VADS Ada 83 compiler. GNAT makes a best effort at duplicating the same semantic interpretation. In particular, `'VADS_Size` applied to a predefined or other primitive type with no Size clause yields the Object_Size (for example, `Natural'Size` is 32 rather than 31 on typical machines). In addition `'VADS_Size` applied to an object gives the result that would be obtained by applying the attribute to the corresponding type.

4.77 Attribute Value_Size

`type'Value_Size` is the number of bits required to represent a value of the given subtype. It is the same as `type'Size`, but, unlike `Size`, may be set for non-first subtypes.

4.78 Attribute Wchar_T_Size

`Standard'Wchar_T_Size` (`Standard` is the only allowed prefix) provides the size in bits of the C `wchar_t` type primarily for constructing the definition of this type in package `Interfaces.C`. The result is a static constant.

4.79 Attribute Word_Size

`Standard'Word_Size` (`Standard` is the only allowed prefix) provides the value `System.Word_Size`. The result is a static constant.

5 Standard and Implementation Defined Restrictions

All Ada Reference Manual-defined Restriction identifiers are implemented:

* language-defined restrictions (see 13.12.1)
* tasking restrictions (see D.7)
* high integrity restrictions (see H.4)

GNAT implements additional restriction identifiers. All restrictions, whether language defined or GNAT-specific, are listed in the following.

5.1 Partition-Wide Restrictions

There are two separate lists of restriction identifiers. The first set requires consistency throughout a partition (in other words, if the restriction identifier is used for any compilation unit in the partition, then all compilation units in the partition must obey the restriction).

5.1.1 Immediate_Reclamation

[RM H.4] This restriction ensures that, except for storage occupied by objects created by allocators and not deallocated via unchecked deallocation, any storage reserved at run time for an object is immediately reclaimed when the object no longer exists.

5.1.2 Max_Asynchronous_Select_Nesting

[RM D.7] Specifies the maximum dynamic nesting level of asynchronous selects. Violations of this restriction with a value of zero are detected at compile time. Violations of this restriction with values other than zero cause Storage_Error to be raised.

5.1.3 Max_Entry_Queue_Length

[RM D.7] This restriction is a declaration that any protected entry compiled in the scope of the restriction has at most the specified number of tasks waiting on the entry at any one time, and so no queue is required. Note that this restriction is checked at run time. Violation of this restriction results in the raising of Program_Error exception at the point of the call.

The restriction `Max_Entry_Queue_Depth` is recognized as a synonym for `Max_Entry_Queue_Length`. This is retained for historical compatibility purposes (and a warning will be generated for its use if warnings on obsolescent features are activated).

5.1.4 Max_Protected_Entries

[RM D.7] Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.

5.1.5 Max_Select_Alternatives

[RM D.7] Specifies the maximum number of alternatives in a selective accept.

5.1.6 Max_Storage_At_Blocking

[RM D.7] Specifies the maximum portion (in storage elements) of a task's Storage_Size that can be retained by a blocked task. A violation of this restriction causes Storage_Error to be raised.

5.1.7 Max_Task_Entries

[RM D.7] Specifies the maximum number of entries per task. The bounds of every entry family of a task unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.

5.1.8 Max_Tasks

[RM D.7] Specifies the maximum number of tasks that may be created, not counting the creation of the environment task. Violations of this restriction with a value of zero are detected at compile time in all contexts. Violations of this restriction with a value greater than zero are detected for library-level tasks at compile time, but are not detected for local tasks at all.

5.1.9 No_Abort_Statements

[RM D.7] There are no abort_statements, and there are no calls to Task_Identification.Abort_Task.

5.1.10 No_Access_Parameter_Allocators

[RM H.4] This restriction ensures at compile time that there are no occurrences of an allocator as the actual parameter to an access parameter.

5.1.11 No_Access_Subprograms

[RM H.4] This restriction ensures at compile time that there are no declarations of access-to-subprogram types.

5.1.12 No_Allocators

[RM H.4] This restriction ensures at compile time that there are no occurrences of an allocator.

5.1.13 No_Anonymous_Allocators

[RM H.4] This restriction ensures at compile time that there are no occurrences of an allocator of anonymous access type.

5.1.14 No_Asynchronous_Control

[RM J.13] This restriction ensures at compile time that there are no semantic dependences on the predefined package Asynchronous_Task_Control.

5.1.15 No_Calendar

[GNAT] This restriction ensures at compile time that there are no semantic dependences on package Calendar.

5.1.16 No_Coextensions

[RM H.4] This restriction ensures at compile time that there are no coextensions. See 3.10.2.

5.1.17 No_Default_Initialization

[GNAT] This restriction prohibits any instance of default initialization of variables or components. The binder implements a consistency check that prevents any unit without the restriction from with'ing a unit with the restriction (this allows the generation of initialization procedures to be skipped, since you can be sure that no call is ever generated to an initialization procedure in a unit with the restriction active). If used in conjunction with Initialize_Scalars or Normalize_Scalars, the effect is to prohibit all cases of variables declared without a specific initializer (including the case of OUT scalar parameters).

5.1.18 No_Delay

[RM H.4] This restriction ensures at compile time that there are no delay statements and no semantic dependences on package Calendar.

5.1.19 No_Dependence

[RM 13.12.1] This restriction ensures at compile time that there are no dependences on a library unit. For GNAT, this includes implicit implementation dependences on units of the runtime library that are created by the compiler to support specific constructs of the language. Here are some examples:

* `System.Arith_64`: 64-bit arithmetics for 32-bit platforms,
* `System.Arith_128`: 128-bit arithmetics for 64-bit platforms,
* `System.Memory`: heap memory allocation routines,
* `System.Memory_Compare`: memory comparison routine (aka `memcmp` for C),
* `System.Memory_Copy`: memory copy routine (aka `memcpy` for C),
* `System.Memory_Move`: memoy move routine (aka `memmove` for C),
* `System.Memory_Set`: memory set routine (aka `memset` for C),
* `System.Stack_Checking[.Operations]`: stack checking without MMU,
* `System.GCC`: support routines from the GCC library.

5.1.20 No_Direct_Boolean_Operators

[GNAT] This restriction ensures that no logical operators (and/or/xor) are used on operands of type Boolean (or any type derived from Boolean). This is intended for use in safety critical programs where the certification protocol requires the use of short-circuit (and then, or else) forms for all composite boolean operations.

5.1.21 No_Dispatch

[RM H.4] This restriction ensures at compile time that there are no occurrences of `T'Class`, for any (tagged) subtype `T`.

5.1.22 No_Dispatching_Calls

[GNAT] This restriction ensures at compile time that the code generated by the compiler involves no dispatching calls. The use of this restriction allows the safe use of record extensions, classwide membership tests and other classwide features not involving implicit dispatching. This restriction ensures that the code contains no indirect calls through a dispatching mechanism. Note that this includes internally-generated calls created by the compiler, for example in the implementation of class-wide objects assignments. The membership test is allowed in the presence of this restriction, because its implementation requires no dispatching. This restriction is comparable to the official Ada restriction `No_Dispatch` except that it is a bit less restrictive in that it allows all classwide constructs that do not imply dispatching. The following example indicates constructs that violate this restriction.

```
package Pkg 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;
  procedure Q (X : DT);
end Pkg;

with Pkg; use Pkg;
procedure Example is
  procedure Test (O : T'Class) is
    N : Natural := O'Size; --  Error: Dispatching call
    C : T'Class := O;      --  Error: implicit Dispatching Call
  begin
    if O in DT'Class then  --  OK   : Membership test
       Q (DT (O));         --  OK   : Type conversion plus direct call
    else
       P (O);              --  Error: Dispatching call
    end if;
  end Test;

  Obj : DT;
begin
  P (Obj);                 --  OK   : Direct call
  P (T (Obj));             --  OK   : Type conversion plus direct call
  P (T'Class (Obj));       --  Error: Dispatching call

  Test (Obj);              --  OK   : Type conversion

  if Obj in T'Class then   --  OK   : Membership test
     null;
  end if;
end Example;
```

5.1.23 No_Dynamic_Attachment

[RM D.7] This restriction ensures that there is no call to any of the operations defined in package Ada.Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

The restriction `No_Dynamic_Interrupts` is recognized as a synonym for `No_Dynamic_Attachment`. This is retained for historical compatibility purposes (and a warning will be generated for its use if warnings on obsolescent features are activated).

5.1.24 No_Dynamic_Priorities

[RM D.7] There are no semantic dependencies on the package Dynamic_Priorities.

5.1.25 No_Entry_Calls_In_Elaboration_Code

[GNAT] This restriction ensures at compile time that no task or protected entry calls are made during elaboration code. As a result of the use of this restriction, the compiler can assume that no code past an accept statement in a task can be executed at elaboration time.

5.1.26 No_Enumeration_Maps

[GNAT] This restriction ensures at compile time that no operations requiring enumeration maps are used (that is Image and Value attributes applied to enumeration types).

5.1.27 No_Exception_Handlers

[GNAT] This restriction ensures at compile time that there are no explicit exception handlers. It also indicates that no exception propagation will be provided. In this mode, exceptions may be raised but will result in an immediate call to the last chance handler, a routine that the user must define with the following profile:

```
procedure Last_Chance_Handler
  (Source_Location : System.Address; Line : Integer);
pragma Export (C, Last_Chance_Handler,
               "__gnat_last_chance_handler");
```

The `Source_Location` parameter is a C null-terminated string representing a message to be associated with the exception (typically the source location of the raise statement generated by the compiler). The Line parameter when nonzero represents the line number in the source program where the raise occurs.

5.1.28 No_Exception_Propagation

[GNAT] This restriction guarantees that exceptions are never propagated to an outer subprogram scope. The only case in which an exception may be raised is when the handler is statically in the same subprogram, so that the effect of a raise is essentially like a goto statement. Any other raise statement (implicit or explicit) will be considered unhandled. Exception handlers are allowed, but may not contain an exception occurrence identifier (exception choice). In addition, use of the package GNAT.Current_Exception is not permitted, and reraise statements (raise with no operand) are not permitted.

5.1.29 No_Exception_Registration

[GNAT] This restriction ensures at compile time that no stream operations for types Exception_Id or Exception_Occurrence are used. This also makes it impossible to pass exceptions to or from a partition with this restriction in a distributed environment. If this restriction is active, the generated code is simplified by omitting the otherwise-required global registration of exceptions when they are declared.

5.1.30 No_Exceptions

[RM H.4] This restriction ensures at compile time that there are no raise statements and no exception handlers and also suppresses the generation of language-defined run-time checks.

5.1.31 No_Finalization

[GNAT] This restriction disables the language features described in chapter 7.6 of the Ada 2005 RM as well as all form of code generation performed by the compiler to support these features. The following types are no longer considered controlled when this restriction is in effect:

* `Ada.Finalization.Controlled`
* `Ada.Finalization.Limited_Controlled`
* Derivations from `Controlled` or `Limited_Controlled`
* Class-wide types
* Protected types
* Task types
* Array and record types with controlled components

The compiler no longer generates code to initialize, finalize or adjust an object or a nested component, either declared on the stack or on the heap. The deallocation of a controlled object no longer finalizes its contents.

5.1.32 No_Fixed_Point

[RM H.4] This restriction ensures at compile time that there are no occurrences of fixed point types and operations.

5.1.33 No_Floating_Point

[RM H.4] This restriction ensures at compile time that there are no occurrences of floating point types and operations.

5.1.34 No_Implicit_Conditionals

[GNAT] This restriction ensures that the generated code does not contain any implicit conditionals, either by modifying the generated code where possible, or by rejecting any construct that would otherwise generate an implicit conditional. Note that this check does not include run time constraint checks, which on some targets may generate implicit conditionals as well. To control the latter, constraint checks can be suppressed in the normal manner. Constructs generating implicit conditionals include comparisons of composite objects and the Max/Min attributes.

5.1.35 No_Implicit_Dynamic_Code

[GNAT] This restriction prevents the compiler from building 'trampolines'. This is a structure that is built on the stack and contains dynamic code to be executed at run time. On some targets, a trampoline is built for the following features: `Access`, `Unrestricted_Access`, or `Address` of a nested subprogram; nested task bodies; primitive operations of nested tagged types. Trampolines do not work on machines that prevent execution of stack data. For example, on windows systems, enabling DEP (data execution protection) will cause trampolines to raise an exception. Trampolines are also quite slow at run time.

On many targets, trampolines have been largely eliminated. Look at the version of system.ads for your target — if it has Always_Compatible_Rep equal to False, then trampolines are largely eliminated. In particular, a trampoline is built for the following features: `Address` of a nested subprogram; `Access` or `Unrestricted_Access` of a nested subprogram, but only if pragma Favor_Top_Level applies, or the access type has a foreign-language convention; primitive operations of nested tagged types.

5.1.36 No_Implicit_Heap_Allocations

[RM D.7] No constructs are allowed to cause implicit heap allocation.

5.1.37 No_Implicit_Protected_Object_Allocations

[GNAT] No constructs are allowed to cause implicit heap allocation of a protected object.

5.1.38 No_Implicit_Task_Allocations

[GNAT] No constructs are allowed to cause implicit heap allocation of a task.

5.1.39 No_Initialize_Scalars

[GNAT] This restriction ensures that no unit in the partition is compiled with pragma Initialize_Scalars. This allows the generation of more efficient code, and in particular eliminates dummy null initialization routines that are otherwise generated for some record and array types.

5.1.40 No_IO

[RM H.4] This restriction ensures at compile time that there are no dependences on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, or Stream_IO.

5.1.41 No_Local_Allocators

[RM H.4] This restriction ensures at compile time that there are no occurrences of an allocator in subprograms, generic subprograms, tasks, and entry bodies.

5.1.42 No_Local_Protected_Objects

[RM D.7] This restriction ensures at compile time that protected objects are only declared at the library level.

5.1.43 No_Local_Tagged_Types

[GNAT] This restriction ensures at compile time that tagged types are only declared at the library level.

5.1.44 No_Local_Timing_Events

[RM D.7] All objects of type Ada.Real_Time.Timing_Events.Timing_Event are declared at the library level.

5.1.45 No_Long_Long_Integers

[GNAT] This partition-wide restriction forbids any explicit reference to type Standard.Long_Long_Integer, and also forbids declaring range types whose implicit base type is Long_Long_Integer, and modular types whose size exceeds Long_Integer'Size.

5.1.46 No_Multiple_Elaboration

[GNAT] When this restriction is active and the static elaboration model is used, and -fpreserve-control-flow is not used, the compiler is allowed to suppress the elaboration counter normally associated with the unit, even if the unit has elaboration code. This counter is typically used to check for access before elaboration and to control multiple elaboration attempts. If the restriction is used, then the situations in which multiple elaboration is possible, including non-Ada main programs and Stand Alone libraries, are not permitted and will be diagnosed by the binder.

5.1.47 No_Nested_Finalization

[RM D.7] All objects requiring finalization are declared at the library level.

5.1.48 No_Protected_Type_Allocators

[RM D.7] This restriction ensures at compile time that there are no allocator expressions that attempt to allocate protected objects.

5.1.49 No_Protected_Types

[RM H.4] This restriction ensures at compile time that there are no declarations of protected types or protected objects.

5.1.50 No_Recursion

[RM H.4] A program execution is erroneous if a subprogram is invoked as part of its execution.

5.1.51 No_Reentrancy

[RM H.4] A program execution is erroneous if a subprogram is executed by two tasks at the same time.

5.1.52 No_Relative_Delay

[RM D.7] This restriction ensures at compile time that there are no delay relative statements and prevents expressions such as `delay 1.23;` from appearing in source code.

5.1.53 No_Requeue_Statements

[RM D.7] This restriction ensures at compile time that no requeue statements are permitted and prevents keyword `requeue` from being used in source code.

The restriction `No_Requeue` is recognized as a synonym for `No_Requeue_Statements`. This is retained for historical compatibility purposes (and a warning will be generated for its use if warnings on oNobsolescent features are activated).

5.1.54 No_Secondary_Stack

[GNAT] This restriction ensures at compile time that the generated code does not contain any reference to the secondary stack. The secondary stack is used to implement functions returning unconstrained objects (arrays or records) on some targets. Suppresses the allocation of secondary stacks for tasks (excluding the environment task) at run time.

5.1.55 No_Select_Statements

[RM D.7] This restriction ensures at compile time no select statements of any kind are permitted, that is the keyword `select` may not appear.

5.1.56 No_Specific_Termination_Handlers

[RM D.7] There are no calls to Ada.Task_Termination.Set_Specific_Handler or to Ada.Task_Termination.Specific_Handler.

5.1.57 No_Specification_of_Aspect

[RM 13.12.1] This restriction checks at compile time that no aspect specification, attribute definition clause, or pragma is given for a given aspect.

5.1.58 No_Standard_Allocators_After_Elaboration

[RM D.7] Specifies that an allocator using a standard storage pool should never be evaluated at run time after the elaboration of the library items of the partition has completed. Otherwise, Storage_Error is raised.

5.1.59 No_Standard_Storage_Pools

[GNAT] This restriction ensures at compile time that no access types use the standard default storage pool. Any access type declared must have an explicit Storage_Pool attribute defined specifying a user-defined storage pool.

5.1.60 No_Stream_Optimizations

[GNAT] This restriction affects the performance of stream operations on types `String`, `Wide_String` and `Wide_Wide_String`. By default, the compiler uses block reads and writes when manipulating `String` objects due to their superior performance. When this restriction is in effect, the compiler performs all IO operations on a per-character basis.

5.1.61 No_Streams

[GNAT] This restriction ensures at compile/bind time that there are no stream objects created and no use of stream attributes. This restriction does not forbid dependences on the package `Ada.Streams`. So it is permissible to with `Ada.Streams` (or another package that does so itself) as long as no actual stream objects are created and no stream attributes are used.

Note that the use of restriction allows optimization of tagged types, since they do not need to worry about dispatching stream operations. To take maximum advantage of this space-saving optimization, any unit declaring a tagged type should be compiled with the restriction, though this is not required.

When pragmas `Discard_Names` and `Restrictions (No_Streams)` simultaneously apply to a tagged type, its Expanded_Name and External_Tag are also initialized with empty strings. In particular, both these pragmas can be applied as configuration pragmas to avoid exposing entity names at binary level for the entire partition.

5.1.62 No_Tagged_Type_Registration

[GNAT] If this restriction is active, then class-wide streaming attributes are not supported. In addition, the subprograms in Ada.Tags are not supported. If this restriction is active, the generated code is simplified by omitting the otherwise-required global registration of tagged types when they are declared. This restriction may be necessary in order to also apply the No_Elaboration_Code restriction.

5.1.63 No_Task_Allocators

[RM D.7] There are no allocators for task types or types containing task subcomponents.

5.1.64 No_Task_At_Interrupt_Priority

[GNAT] This restriction ensures at compile time that there is no Interrupt_Priority aspect or pragma for a task or a task type. As a consequence, the tasks are always created with a priority below that an interrupt priority.

5.1.65 No_Task_Attributes_Package

[GNAT] This restriction ensures at compile time that there are no implicit or explicit dependencies on the package `Ada.Task_Attributes`.

The restriction `No_Task_Attributes` is recognized as a synonym for `No_Task_Attributes_Package`. This is retained for historical compatibility purposes (and a warning will be generated for its use if warnings on obsolescent features are activated).

5.1.66 No_Task_Hierarchy

[RM D.7] All (non-environment) tasks depend directly on the environment task of the partition.

5.1.67 No_Task_Termination

[RM D.7] Tasks that terminate are erroneous.

5.1.68 No_Tasking

[GNAT] This restriction prevents the declaration of tasks or task types throughout the partition. It is similar in effect to the use of `Max_Tasks => 0` except that violations are caught at compile time and cause an error message to be output either by the compiler or binder.

5.1.69 No_Terminate_Alternatives

[RM D.7] There are no selective accepts with terminate alternatives.

5.1.70 No_Unchecked_Access

[RM H.4] This restriction ensures at compile time that there are no occurrences of the Unchecked_Access attribute.

5.1.71 No_Unchecked_Conversion

[RM J.13] This restriction ensures at compile time that there are no semantic dependences on the predefined generic function Unchecked_Conversion.

5.1.72 No_Unchecked_Deallocation

[RM J.13] This restriction ensures at compile time that there are no semantic dependences on the predefined generic procedure Unchecked_Deallocation.

5.1.73 No_Use_Of_Attribute

[RM 13.12.1] This is a standard Ada 2012 restriction that is GNAT defined in earlier versions of Ada.

5.1.74 No_Use_Of_Entity

[GNAT] This restriction ensures at compile time that there are no references to the entity given in the form

```
No_Use_Of_Entity => Name
```

where `Name` is the fully qualified entity, for example

```
No_Use_Of_Entity => Ada.Text_IO.Put_Line
```

5.1.75 No_Use_Of_Pragma

[RM 13.12.1] This is a standard Ada 2012 restriction that is GNAT defined in earlier versions of Ada.

5.1.76 Pure_Barriers

[GNAT] This restriction ensures at compile time that protected entry barriers are restricted to:

* components of the protected object (excluding selection from dereferences),
* constant declarations,
* named numbers,
* enumeration literals,
* integer literals,
* real literals,
* character literals,
* implicitly defined comparison operators,
* uses of the Standard."not" operator,
* short-circuit operator,
* the Count attribute

This restriction is a relaxation of the Simple_Barriers restriction, but still ensures absence of side effects, exceptions, and recursion during the evaluation of the barriers.

5.1.77 Simple_Barriers

[RM D.7] This restriction ensures at compile time that barriers in entry declarations for protected types are restricted to either static boolean expressions or references to simple boolean variables defined in the private part of the protected type. No other form of entry barriers is permitted.

The restriction `Boolean_Entry_Barriers` is recognized as a synonym for `Simple_Barriers`. This is retained for historical compatibility purposes (and a warning will be generated for its use if warnings on obsolescent features are activated).

5.1.78 Static_Priorities

[GNAT] This restriction ensures at compile time that all priority expressions are static, and that there are no dependences on the package `Ada.Dynamic_Priorities`.

5.1.79 Static_Storage_Size

[GNAT] This restriction ensures at compile time that any expression appearing in a Storage_Size pragma or attribute definition clause is static.

5.2 Program Unit Level Restrictions

The second set of restriction identifiers does not require partition-wide consistency. The restriction may be enforced for a single compilation unit without any effect on any of the other compilation units in the partition.

5.2.1 No_Elaboration_Code

[GNAT] This restriction ensures at compile time that no elaboration code is generated. Note that this is not the same condition as is enforced by pragma `Preelaborate`. There are cases in which pragma `Preelaborate` still permits code to be generated (e.g., code to initialize a large array to all zeroes), and there are cases of units which do not meet the requirements for pragma `Preelaborate`, but for which no elaboration code is generated. Generally, it is the case that preelaborable units will meet the restrictions, with the exception of large aggregates initialized with an others_clause, and exception declarations (which generate calls to a run-time registry procedure). This restriction is enforced on a unit by unit basis, it need not be obeyed consistently throughout a partition.

In the case of aggregates with others, if the aggregate has a dynamic size, there is no way to eliminate the elaboration code (such dynamic bounds would be incompatible with `Preelaborate` in any case). If the bounds are static, then use of this restriction actually modifies the code choice of the compiler to avoid generating a loop, and instead generate the aggregate statically if possible, no matter how many times the data for the others clause must be repeatedly generated.

It is not possible to precisely document the constructs which are compatible with this restriction, since, unlike most other restrictions, this is not a restriction on the source code, but a restriction on the generated object code. For example, if the source contains a declaration:

```
Val : constant Integer := X;
```

where X is not a static constant, it may be possible, depending on complex optimization circuitry, for the compiler to figure out the value of X at compile time, in which case this initialization can be done by the loader, and requires no initialization code. It is not possible to document the precise conditions under which the optimizer can figure this out.

Note that this the implementation of this restriction requires full code generation. If it is used in conjunction with "semantics only" checking, then some cases of violations may be missed.

When this restriction is active, we are not requesting control-flow preservation with -fpreserve-control-flow, and the static elaboration model is used, the compiler is allowed to suppress the elaboration counter normally associated with the unit. This counter is typically used to check for access before elaboration and to control multiple elaboration attempts.

5.2.2 No_Dynamic_Accessibility_Checks

[GNAT] No dynamic accessibility checks are generated when this restriction is in effect. Instead, dangling references are prevented via more conservative compile-time checking. More specifically, existing compile-time checks are enforced but with more conservative assumptions about the accessibility levels of the relevant entities. These conservative assumptions eliminate the need for dynamic accessibility checks.

These new rules for computing (at compile-time) the accessibility level of an anonymous access type T are as follows:

* If T is a function result type then, from the caller's perspective, its level is that of the innermost master enclosing the function call. From the callee's perspective, the level of parameters and local variables of the callee is statically deeper than the level of T.

  For any other accessibility level L such that the level of parameters and local variables of the callee is statically deeper than L, the level of T (from the callee's perspective) is also statically deeper than L.
* If T is the type of a formal parameter then, from the caller's perspective, its level is at least as deep as that of the type of the corresponding actual parameter (whatever that actual parameter might be). From the callee's perspective, the level of parameters and local variables of the callee is statically deeper than the level of T.
* If T is the type of a discriminant then its level is that of the discriminated type.
* If T is the type of a stand-alone object then its level is the level of the object.
* In all other cases, the level of T is as defined by the existing rules of Ada.

5.2.3 No_Dynamic_Sized_Objects

[GNAT] This restriction disallows certain constructs that might lead to the creation of dynamic-sized composite objects (or array or discriminated type). An array subtype indication is illegal if the bounds are not static or references to discriminants of an enclosing type. A discriminated subtype indication is illegal if the type has discriminant-dependent array components or a variant part, and the discriminants are not static. In addition, array and record aggregates are illegal in corresponding cases. Note that this restriction does not forbid access discriminants. It is often a good idea to combine this restriction with No_Secondary_Stack.

5.2.4 No_Entry_Queue

[GNAT] This restriction is a declaration that any protected entry compiled in the scope of the restriction has at most one task waiting on the entry at any one time, and so no queue is required. This restriction is not checked at compile time. A program execution is erroneous if an attempt is made to queue a second task on such an entry.

5.2.5 No_Implementation_Aspect_Specifications

[RM 13.12.1] This restriction checks at compile time that no GNAT-defined aspects are present. With this restriction, the only aspects that can be used are those defined in the Ada Reference Manual.

5.2.6 No_Implementation_Attributes

[RM 13.12.1] This restriction checks at compile time that no GNAT-defined attributes are present. With this restriction, the only attributes that can be used are those defined in the Ada Reference Manual.

5.2.7 No_Implementation_Identifiers

[RM 13.12.1] This restriction checks at compile time that no implementation-defined identifiers (marked with pragma Implementation_Defined) occur within language-defined packages.

5.2.8 No_Implementation_Pragmas

[RM 13.12.1] This restriction checks at compile time that no GNAT-defined pragmas are present. With this restriction, the only pragmas that can be used are those defined in the Ada Reference Manual.

5.2.9 No_Implementation_Restrictions

[GNAT] This restriction checks at compile time that no GNAT-defined restriction identifiers (other than `No_Implementation_Restrictions` itself) are present. With this restriction, the only other restriction identifiers that can be used are those defined in the Ada Reference Manual.

5.2.10 No_Implementation_Units

[RM 13.12.1] This restriction checks at compile time that there is no mention in the context clause of any implementation-defined descendants of packages Ada, Interfaces, or System.

5.2.11 No_Implicit_Aliasing

[GNAT] This restriction, which is not required to be partition-wide consistent, requires an explicit aliased keyword for an object to which 'Access, 'Unchecked_Access, or 'Address is applied, and forbids entirely the use of the 'Unrestricted_Access attribute for objects. Note: the reason that Unrestricted_Access is forbidden is that it would require the prefix to be aliased, and in such cases, it can always be replaced by the standard attribute Unchecked_Access which is preferable.

5.2.12 No_Implicit_Loops

[GNAT] This restriction ensures that the generated code of the unit marked with this restriction does not contain any implicit `for` loops, either by modifying the generated code where possible, or by rejecting any construct that would otherwise generate an implicit `for` loop. If this restriction is active, it is possible to build large array aggregates with all static components without generating an intermediate temporary, and without generating a loop to initialize individual components. Otherwise, a loop is created for arrays larger than about 5000 scalar components. Note that if this restriction is set in the spec of a package, it will not apply to its body.

5.2.13 No_Obsolescent_Features

[RM 13.12.1] This restriction checks at compile time that no obsolescent features are used, as defined in Annex J of the Ada Reference Manual.

5.2.14 No_Wide_Characters

[GNAT] This restriction ensures at compile time that no uses of the types `Wide_Character` or `Wide_String` or corresponding wide wide types appear, and that no wide or wide wide string or character literals appear in the program (that is literals representing characters not in type `Character`).

5.2.15 Static_Dispatch_Tables

[GNAT] This restriction checks at compile time that all the artifacts associated with dispatch tables can be placed in read-only memory.

5.2.16 SPARK_05

[GNAT] This restriction no longer has any effect and is superseded by SPARK 2014, whose restrictions are checked by the tool GNATprove. To check that a codebase respects SPARK 2014 restrictions, mark the code with pragma or aspect `SPARK_Mode`, and run the tool GNATprove at Stone assurance level, as follows:

```
gnatprove -P project.gpr --mode=stone
```

or equivalently:

```
gnatprove -P project.gpr --mode=check_all
```

6 Implementation Advice

The main text of the Ada Reference Manual describes the required behavior of all Ada compilers, and the GNAT compiler conforms to these requirements.

In addition, there are sections throughout the Ada Reference Manual headed by the phrase 'Implementation advice'. These sections are not normative, i.e., they do not specify requirements that all compilers must follow. Rather they provide advice on generally desirable behavior. They are not requirements, because they describe behavior that cannot be provided on all systems, or may be undesirable on some systems.

As far as practical, GNAT follows the implementation advice in the Ada Reference Manual. Each such RM section corresponds to a section in this chapter whose title specifies the RM section number and paragraph number and the subject of the advice. The contents of each section consists of the RM text within quotation marks, followed by the GNAT interpretation of the advice. Most often, this simply says 'followed', which means that GNAT follows the advice. However, in a number of cases, GNAT deliberately deviates from this advice, in which case the text describes what GNAT does and why.

6.1 RM 1.1.3(20): Error Detection

> "If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise `Program_Error` if feasible."

Not relevant. All specialized needs annex features are either supported, or diagnosed at compile time.

6.2 RM 1.1.3(31): Child Units

> "If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit."

Followed.

6.3 RM 1.1.5(12): Bounded Errors

> "If an implementation detects a bounded error or erroneous execution, it should raise `Program_Error`."

Followed in all cases in which the implementation detects a bounded error or erroneous execution. Not all such situations are detected at runtime.

6.4 RM 2.8(16): Pragmas

> "Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics."

The following implementation defined pragmas are exceptions to this rule:

Pragma	**Explanation**
‘Abort_Defer’	Affects semantics
‘Ada_83’	Affects legality
‘Assert’	Affects semantics
‘CPP_Class’	Affects semantics
‘CPP_Constructor’	Affects semantics
‘Debug’	Affects semantics
‘Interface_Name’	Affects semantics
‘Machine_Attribute’	Affects semantics
‘Unimplemented_Unit’	Affects legality
‘Unchecked_Union’	Affects semantics

In each of the above cases, it is essential to the purpose of the pragma that this advice not be followed. For details see [Implementation Defined Pragmas], page 4.

6.5 RM 2.8(17-19): Pragmas

> “Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows:
>
> * A pragma used to complete a declaration, such as a pragma `Import`;
> * A pragma used to configure the environment by adding, removing, or replacing `library_items`.”

See [RM 2.8(16); Pragmas], page 160.

6.6 RM 3.5.2(5): Alternative Character Sets

> “If an implementation supports a mode with alternative interpretations for `Character` and `Wide_Character`, the set of graphic characters of `Character` should nevertheless remain a proper subset of the set of graphic characters of `Wide_Character`. Any character set ‘localizations’ should be reflected in the results of the subprograms defined in the language-defined package `Characters.Handling` (see A.3) available in such a mode. In a mode with an alternative interpretation of `Character`, the implementation should also support a corresponding change in what is a legal `identifier_letter`.”

Not all wide character modes follow this advice, in particular the JIS and IEC modes reflect standard usage in Japan, and in these encoding, the upper half of the Latin-1 set is not part of the wide-character subset, since the most significant bit is used for wide character encoding. However, this only applies to the external forms. Internally there is no such restriction.

6.7 RM 3.5.4(28): Integer Types

> "An implementation should support `Long_Integer` in addition to `Integer` if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package `Standard`. Instead, appropriate named integer subtypes should be provided in the library package `Interfaces` (see B.2)."

`Long_Integer` is supported. Other standard integer types are supported so this advice is not fully followed. These types are supported for convenient interface to C, and so that all hardware types of the machine are easily available.

6.8 RM 3.5.4(29): Integer Types

> "An implementation for a two's complement machine should support modular types with a binary modulus up to `System.Max_Int*2+2`. An implementation should support a non-binary modules up to `Integer'Last`."

Followed.

6.9 RM 3.5.5(8): Enumeration Values

> "For the evaluation of a call on `S'Pos` for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type (perhaps due to an un-initialized variable), then the implementation should raise `Program_Error`. This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_clause."

Followed.

6.10 RM 3.5.7(17): Float Types

> "An implementation should support `Long_Float` in addition to `Float` if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package `Standard`. Instead, appropriate named floating point subtypes should be provided in the library package `Interfaces` (see B.2)."

`Short_Float` and `Long_Long_Float` are also provided. The former provides improved compatibility with other implementations supporting this type. The latter corresponds to the highest precision floating-point type supported by the hardware. On most machines, this will be the same as `Long_Float`, but on some machines, it will correspond to the IEEE extended form. The notable case is all x86 implementations, where `Long_Long_Float` corresponds to the 80-bit extended precision format supported in hardware on this processor.

Note that the 128-bit format on SPARC is not supported, since this is a software rather than a hardware format.

6.11 RM 3.6.2(11): Multidimensional Arrays

> "An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). However, if a pragma `Convention` (`Fortran`, ...) applies to a multidimensional array type, then column-major order should be used instead (see B.5, 'Interfacing with Fortran')."

Followed.

6.12 RM 9.6(30-31): Duration'Small

> "Whenever possible in an implementation, the value of `Duration'Small` should be no greater than 100 microseconds."

Followed. (`Duration'Small` = 10**(-9)).

> "The time base for `delay_relative_statements` should be monotonic; it need not be the same time base as used for `Calendar.Clock`."

Followed.

6.13 RM 10.2.1(12): Consistent Representation

> "In an implementation, a type declared in a pre-elaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version."

Followed, except in the case of tagged types. Tagged types involve implicit pointers to a local copy of a dispatch table, and these pointers have representations which thus depend on a particular elaboration of the package. It is not easy to see how it would be possible to follow this advice without severely impacting efficiency of execution.

6.14 RM 11.4.1(19): Exception Information

> "`Exception_Message` by default and `Exception_Information` should produce information useful for debugging. `Exception_Message` should be short, about one line. `Exception_Information` can be long. `Exception_Message` should not include the `Exception_Name`. `Exception_Information` should include both the `Exception_Name` and the `Exception_Message`."

Followed. For each exception that doesn't have a specified `Exception_Message`, the compiler generates one containing the location of the raise statement. This location has the form 'file_name:line', where file_name is the short file name (without path information) and line is the line number in the file. Note that in the case of the Zero Cost Exception mechanism, these messages become redundant with the Exception_Information that contains a full backtrace of the calling sequence, so they are disabled. To disable explicitly the generation of the source location message, use the Pragma `Discard_Names`.

6.15 RM 11.5(28): Suppression of Checks

> "The implementation should minimize the code executed for checks that have been suppressed."

Followed.

6.16 RM 13.1 (21-24): Representation Clauses

> "The recommended level of support for all representation items is qualified as follows:
>
> An implementation need not support representation items containing nonstatic expressions, except that an implementation should support a representation item for a given entity if each nonstatic expression in the representation item is a name that statically denotes a constant declared before the entity."

Followed. In fact, GNAT goes beyond the recommended level of support by allowing nonstatic expressions in some representation clauses even without the need to declare constants initialized with the values of such expressions. For example:

```
X : Integer;
Y : Float;
for Y'Address use X'Address;
```

is accepted directly by GNAT.

> "An implementation need not support a specification for the `Size` for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints."

Followed. Size Clauses are not permitted on nonstatic components, as described above.

> "An aliased component, or a component whose type is by-reference, should always be allocated at an addressable location."

Followed.

6.17 RM 13.2(6-8): Packed Types

> "If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.
>
> The recommended level of support pragma `Pack` is:
>
> For a packed record type, the components should be packed as tightly as possible subject to the Sizes of the component subtypes, and subject to any 'record_representation_clause' that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose `Size` is greater than the word size may be allocated an integral number of words."

Followed. Tight packing of arrays is supported for all component sizes up to 64-bits. If the array component size is 1 (that is to say, if the component is a boolean type or an enumeration type with two values) then values of the type are implicitly initialized to zero. This happens both for objects of the packed type, and for objects that have a subcomponent of the packed type.

6.18 RM 13.3(14-19): Address Clauses

> "For an array `X`, `X'Address` should point at the first component of the array, and not at the array bounds."

Followed.

> "The recommended level of support for the `Address` attribute is:
>
> `X'Address` should produce a useful result if `X` is an object that is aliased or of a by-reference type, or is an entity whose `Address` has been specified."

Followed. A valid address will be produced even if none of those conditions have been met. If necessary, the object is forced into memory to ensure the address is valid.

> "An implementation should support `Address` clauses for imported subprograms."

Followed.

> "Objects (including subcomponents) that are aliased or of a by-reference type should be allocated on storage element boundaries."

Followed.

> "If the `Address` of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases."

Followed.

6.19 RM 13.3(29-35): Alignment Clauses

> "The recommended level of support for the `Alignment` attribute for subtypes is:
>
> An implementation should support specified Alignments that are factors and multiples of the number of storage elements per word, subject to the following:"

Followed.

> "An implementation need not support specified Alignments for combinations of Sizes and Alignments that cannot be easily loaded and stored by available machine instructions."

Followed.

> "An implementation need not support specified Alignments that are greater than the maximum `Alignment` the implementation ever returns by default."

Followed.

> "The recommended level of support for the `Alignment` attribute for objects is:
>
> Same as above, for subtypes, but in addition:"

Followed.

> "For stand-alone library-level objects of statically constrained subtypes, the implementation should support all alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes."

Followed.

6.20 RM 13.3(42-43): Size Clauses

> "The recommended level of support for the `Size` attribute of objects is:
>
> A `Size` clause should be supported for an object if the specified `Size` is at least as large as its subtype's `Size`, and corresponds to a size in storage elements that is a multiple of the object's `Alignment` (if the `Alignment` is nonzero)."

Followed.

6.21 RM 13.3(50-56): Size Clauses

> "If the `Size` of a subtype is specified, and allows for efficient independent addressability (see 9.10) on the target architecture, then the `Size` of the following objects of the subtype should equal the `Size` of the subtype:
>
> Aliased objects (including components)."

Followed.

> "*Size* clause on a composite subtype should not affect the internal layout of components."

Followed. But note that this can be overridden by use of the implementation pragma Implicit_Packing in the case of packed arrays.

> "The recommended level of support for the `Size` attribute of subtypes is:
>
> The `Size` (if not specified) of a static discrete or fixed point subtype should be the number of bits needed to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified `Size` for it that reflects this representation."

Followed.

> "For a subtype implemented with levels of indirection, the `Size` should include the size of the pointers, but not the size of what they point at."

Followed.

6.22 RM 13.3(71-73): Component Size Clauses

> "The recommended level of support for the `Component_Size` attribute is:
>
> An implementation need not support specified `Component_Sizes` that are less than the `Size` of the component subtype."

Followed.

> "An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps between components. For other Component_Sizes (if supported), the array should contain no gaps between components when packing is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation."

Followed.

6.23 RM 13.4(9-10): Enumeration Representation Clauses

> "The recommended level of support for enumeration representation clauses is:
>
> An implementation need not support enumeration representation clauses for boolean types, but should at minimum support the internal codes in the range `System.Min_Int .. System.Max_Int`."

Followed.

6.24 RM 13.5.1(17-22): Record Representation Clauses

> "The recommended level of support for 'record_representation_clause's is:
>
> An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model."

Followed.

> "A storage place should be supported if its size is equal to the `Size` of the component subtype, and it starts and ends on a boundary that obeys the `Alignment` of the component subtype."

Followed.

> "If the default bit ordering applies to the declaration of a given type, then for a component whose subtype's `Size` is less than the word size, any storage place that does not cross an aligned word boundary should be supported."

Followed.

> "An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place."

Followed. The storage place for the tag field is the beginning of the tagged record, and its size is Address'Size. GNAT will reject an explicit component clause for the tag field.

> "An implementation need not support a 'component_clause' for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified."

Followed. The above advice on record representation clauses is followed, and all mentioned features are implemented.

6.25 RM 13.5.2(5): Storage Place Attributes

> "If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontinuously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes."

Followed. There are no such components in GNAT.

6.26 RM 13.5.3(7-8): Bit Ordering

> "The recommended level of support for the non-default bit ordering is:
>
> The implementation should support the nondefault bit ordering in addition to the default bit ordering."

Followed.

6.27 RM 13.7(37): Address as Private

> "*Address* should be of a private type."

Followed.

6.28 RM 13.7.1(16): Address Operations

> "Operations in `System` and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to 'wrap around'. Operations that do not make sense should raise `Program_Error`."

Followed. Address arithmetic is modular arithmetic that wraps around. No operation raises `Program_Error`, since all operations make sense.

6.29 RM 13.9(14-17): Unchecked Conversion

> "The `Size` of an array object should not include its bounds; hence, the bounds should not be part of the converted data."

Followed.

> "The implementation should not generate unnecessary run-time checks to ensure that the representation of `S` is a representation of the target

> type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment."

Followed. There are no restrictions on unchecked conversion. A warning is generated if the source and target types do not have the same size since the semantics in this case may be target dependent.

> "The recommended level of support for unchecked conversions is:
>
> Unchecked conversions should be supported and should be reversible in the cases where this clause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph."

Followed.

6.30 RM 13.11(23-25): Implicit Heap Usage

> "An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator."

Followed, the only other points at which heap storage is dynamically allocated are as follows:

* At initial elaboration time, to allocate dynamically sized global objects.
* To allocate space for a task when a task is created.
* To extend the secondary stack dynamically when needed. The secondary stack is used for returning variable length results.

> "A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects."

Followed.

> "A storage pool for an anonymous access type should be created at the point of an allocator for the type, and be reclaimed when the designated object becomes inaccessible."

Followed.

6.31 RM 13.11.2(17): Unchecked Deallocation

> "For a standard storage pool, `Free` should actually reclaim the storage."

Followed.

6.32 RM 13.13.2(1.6): Stream Oriented Attributes

> "If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the

> nearest factor or multiple of the word size that is also a multiple of the stream element size."

Followed, except that the number of stream elements is 1, 2, 3, 4 or 8. The Stream_Size may be used to override the default choice.

The default implementation is based on direct binary representations and is therefore target- and endianness-dependent. To address this issue, GNAT also supplies an alternate implementation of the stream attributes `Read` and `Write`, which uses the target-independent XDR standard representation for scalar types. This XDR alternative can be enabled via the binder switch -xdr.

6.33 RM A.1(52): Names of Predefined Numeric Types

> "If an implementation provides additional named predefined integer types, then the names should end with `Integer` as in `Long_Integer`. If an implementation provides additional named predefined floating point types, then the names should end with `Float` as in `Long_Float`."

Followed.

6.34 RM A.3.2(49): `Ada.Characters.Handling`

> "If an implementation provides a localized definition of `Character` or `Wide_Character`, then the effects of the subprograms in `Characters.Handling` should reflect the localizations. See also 3.5.2."

Followed. GNAT provides no such localized definitions.

6.35 RM A.4.4(106): Bounded-Length String Handling

> "Bounded string objects should not be implemented by implicit pointers and dynamic allocation."

Followed. No implicit pointers or dynamic allocation are used.

6.36 RM A.5.2(46-47): Random Number Generation

> "Any storage associated with an object of type `Generator` should be reclaimed on exit from the scope of the object."

Followed.

> "If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of `Initiator` passed to `Reset` should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value."

Followed. The generator period is sufficiently long for the first condition here to hold true.

6.37 RM A.10.7(23): `Get_Immediate`

> "The `Get_Immediate` procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be available if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of `Get_Immediate`."

Followed on all targets except VxWorks. For VxWorks, there is no way to provide this functionality that does not result in the input buffer being flushed before the `Get_Immediate` call. A special unit `Interfaces.Vxworks.IO` is provided that contains routines to enable this functionality.

6.38 RM A.18: `Containers`

All implementation advice pertaining to Ada.Containers and its child units (that is, all implementation advice occurring within section A.18 and its subsections) is followed except for A.18.24(17):

> "Bounded ordered set objects should be implemented without implicit pointers or dynamic allocation. "

The implementations of the two Reference_Preserving_Key functions of the generic package Ada.Containers.Bounded_Ordered_Sets each currently make use of dynamic allocation; other operations on bounded ordered set objects follow the implementation advice.

6.39 RM B.1(39-41): Pragma `Export`

> "If an implementation supports pragma `Export` to a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms whose link names are `adainit` and `adafinal`. `adainit` should contain the elaboration code for library units. `adafinal` should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called."

Followed.

> "Automatic elaboration of pre-elaborated packages should be provided when pragma `Export` is supported."

Followed when the main program is in Ada. If the main program is in a foreign language, then `adainit` must be called to elaborate pre-elaborated packages.

> "For each supported convention 'L' other than `Intrinsic`, an implementation should support `Import` and `Export` pragmas for objects of 'L'-compatible types and for subprograms, and pragma *Convention* for 'L'-eligible types and for subprograms, presuming the other language has corresponding features. Pragma `Convention` need not be supported for scalar types."

Followed.

6.40 RM B.2(12-13): Package `Interfaces`

> "For each implementation-defined convention identifier, there should be a child package of package Interfaces with the corresponding name. This package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in `Interfaces`."

Followed.

> "An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following clauses."

Followed. GNAT provides all the packages described in this section.

6.41 RM B.3(63-71): Interfacing with C

> "An implementation should support the following interface correspondences between Ada and C."

Followed.

> "An Ada procedure corresponds to a void-returning C function."

Followed.

> "An Ada function corresponds to a non-void C function."

Followed.

> "An Ada `in` scalar parameter is passed as a scalar argument to a C function."

Followed.

> "An Ada `in` parameter of an access-to-object type with designated type `T` is passed as a `t*` argument to a C function, where `t` is the C type corresponding to the Ada type `T`."

Followed.

> "An Ada access `T` parameter, or an Ada `out` or `in out` parameter of an elementary type `T`, is passed as a `t*` argument to a C function, where `t` is the C type corresponding to the Ada type `T`. In the case of an elementary `out` or `in out` parameter, a pointer to a temporary copy is used to preserve by-copy semantics."

Followed.

> "An Ada parameter of a record type `T`, of any mode, is passed as a `t*` argument to a C function, where `t` is the C structure corresponding to the Ada type `T`."

Followed. This convention may be overridden by the use of the C_Pass_By_Copy pragma, or Convention, or by explicitly specifying the mechanism for a given call using an extended import or export pragma.

> "An Ada parameter of an array type with component type `T`, of any mode, is passed as a `t*` argument to a C function, where `t` is the C type corresponding to the Ada type `T`."

Followed.

> "An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification."

Followed.

6.42 RM B.4(95-98): Interfacing with COBOL

> "An Ada implementation should support the following interface correspondences between Ada and COBOL."

Followed.

> "An Ada access `T` parameter is passed as a `BY REFERENCE` data item of the COBOL type corresponding to `T`."

Followed.

> "An Ada in scalar parameter is passed as a `BY CONTENT` data item of the corresponding COBOL type."

Followed.

> "Any other Ada parameter is passed as a `BY REFERENCE` data item of the COBOL type corresponding to the Ada parameter type; for scalars, a local copy is used if necessary to ensure by-copy semantics."

Followed.

6.43 RM B.5(22-26): Interfacing with Fortran

> "An Ada implementation should support the following interface correspondences between Ada and Fortran:"

Followed.

> "An Ada procedure corresponds to a Fortran subroutine."

Followed.

> "An Ada function corresponds to a Fortran function."

Followed.

> "An Ada parameter of an elementary, array, or record type `T` is passed as a `T` argument to a Fortran procedure, where `T` is the Fortran type corresponding to the Ada type `T`, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics."

Followed.

> "An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification."

Followed.

6.44 RM C.1(3-5): Access to Machine Operations

> "The machine code or intrinsic support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any."

Followed.

> "The interfacing pragmas (see Annex B) should support interface to assembler; the default assembler should be associated with the convention identifier `Assembler`."

Followed.

> "If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported."

Followed.

6.45 RM C.1(10-16): Access to Machine Operations

> "The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms."

Followed for both intrinsics and machine-code subprograms.

> "It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs."

Followed. A full set of machine operation intrinsic subprograms is provided.

> "Atomic read-modify-write operations—e.g., test and set, compare and swap, decrement and test, enqueue/dequeue."

Followed on any target supporting such operations.

> "Standard numeric functions—e.g.:, sin, log."

Followed on any target supporting such operations.

> "String manipulation operations—e.g.:, translate and test."

Followed on any target supporting such operations.

> "Vector operations—e.g.:, compare vector against thresholds."

Followed on any target supporting such operations.

> "Direct operations on I/O ports."

Followed on any target supporting such operations.

6.46 RM C.3(28): Interrupt Support

> "If the `Ceiling_Locking` policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for a finer-grain control of interrupt blocking."

Followed. The underlying system does not allow for finer-grain control of interrupt blocking.

6.47 RM C.3.1(20-21): Protected Procedure Handlers

> "Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware."

Followed on any target where the underlying operating system permits such direct calls.

> "Whenever practical, violations of any implementation-defined restrictions should be detected before run time."

Followed. Compile time warnings are given when possible.

6.48 RM C.3.2(25): Package `Interrupts`

> "If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to `Parameterless_Handler` should be specified in a child package of `Interrupts`, with the same operations as in the predefined package Interrupts."

Followed.

6.49 RM C.4(14): Pre-elaboration Requirements

> "It is recommended that pre-elaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements."

Followed. Executable code is generated in some cases, e.g., loops to initialize large arrays.

6.50 RM C.5(8): Pragma `Discard_Names`

> "If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity."

Followed.

6.51 RM C.7.2(30): The Package Task_Attributes

> "Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the task's attributes,

> or by using the pre-allocated storage for the first `N` attribute objects, and the heap for the others. In the latter case, `N` should be documented."

Not followed. This implementation is not targeted to such a domain.

6.52 RM D.3(17): Locking Policies

> "The implementation should use names that end with `_Locking` for locking policies defined by the implementation."

Followed. Two implementation-defined locking policies are defined, whose names (`Inheritance_Locking` and `Concurrent_Readers_Locking`) follow this suggestion.

6.53 RM D.4(16): Entry Queuing Policies

> "Names that end with `_Queuing` should be used for all implementation-defined queuing policies."

Followed. No such implementation-defined queuing policies exist.

6.54 RM D.6(9-10): Preemptive Abort

> "Even though the 'abort_statement' is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the 'abort_statement' to block."

Followed.

> "On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this."

Followed.

6.55 RM D.7(21): Tasking Restrictions

> "When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation."

GNAT currently takes advantage of these restrictions by providing an optimized run time when the Ravenscar profile and the GNAT restricted run time set of restrictions are specified. See pragma `Profile (Ravenscar)` and pragma `Profile (Restricted)` for more details.

6.56 RM D.8(47-49): Monotonic Time

> "When appropriate, implementations should provide configuration mechanisms to change the value of `Tick`."

Such configuration mechanisms are not appropriate to this implementation and are thus not supported.

> "It is recommended that `Calendar.Clock` and `Real_Time.Clock` be implemented as transformations of the same time base."

Followed.

> "It is recommended that the best time base which exists in the underlying system be available to the application through `Clock`. *Best* may mean highest accuracy or largest range."

Followed.

6.57 RM E.5(28-29): Partition Communication Subsystem

> "Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC-receiver with different messages and should allow them to block until the corresponding subprogram body returns."

A separately supplied PCS that can be used with GNAT when combined with the PolyORB product (NB! See the note in [PolyORB], page 372, regarding the lifetime of this product).

> "The `Write` operation on a stream of type `Params_Stream_Type` should raise `Storage_Error` if it runs out of space trying to write the `Item` into the stream."

6.58 RM F(7): COBOL Support

> "If COBOL (respectively, C) is widely supported in the target environment, implementations supporting the Information Systems Annex should provide the child package `Interfaces.COBOL` (respectively, `Interfaces.C`) specified in Annex B and should support a `convention_identifier` of COBOL (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language."

Followed.

6.59 RM F.1(2): Decimal Radix Support

> "Packed decimal should be used as the internal representation for objects of subtype `S` when `S`'Machine_Radix = 10."

Not followed. GNAT ignores `S`'Machine_Radix and always uses binary representations.

6.60 RM G: Numerics

> "If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package `Interfaces.Fortran` (respectively, `Interfaces.C`) specified in Annex B and should support a `convention_identifier` of Fortran (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language."

Followed.

6.61 RM G.1.1(56-58): Complex Types

> "Because the usual mathematical meaning of multiplication of a complex operand and a real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to IEC 559:1989, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand."

Not followed.

> "Similarly, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex addition. In implementations in which the `Signed_Zeros` attribute of the component type is `True` (and which therefore conform to IEC 559:1989 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand."

Not followed.

> "Implementations in which `Real'Signed_Zeros` is `True` should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the `Argument` function should have the sign of the imaginary component of the parameter `X` when the point represented by that parameter lies on the positive real axis; as another, the sign of the imaginary component of the `Compose_From_Polar` function should be the same as (respectively, the opposite of) that of the `Argument` parameter when that parameter has a value of zero and the `Modulus` parameter has a nonnegative (respectively, negative) value."

Followed.

6.62 RM G.1.2(49): Complex Elementary Functions

> "Implementations in which `Complex_Types.Real'Signed_Zeros` is `True` should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary

> functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative."

Followed.

6.63 RM G.2.4(19): Accuracy Requirements

> "The versions of the forward trigonometric functions without a `Cycle` parameter should not be implemented by calling the corresponding version with a `Cycle` parameter of `2.0*Numerics.Pi`, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of `Log` without a `Base` parameter should not be implemented by calling the corresponding version with a `Base` parameter of `Numerics.e`."

Followed.

6.64 RM G.2.6(15): Complex Arithmetic Accuracy

> "The version of the `Compose_From_Polar` function without a `Cycle` parameter should not be implemented by calling the corresponding version with a `Cycle` parameter of `2.0*Numerics.Pi`, since this will not provide the required accuracy in some portions of the domain."

Followed.

6.65 RM H.6(15/2): Pragma Partition_Elaboration_Policy

> "If the partition elaboration policy is `Sequential` and the Environment task becomes permanently blocked during elaboration then the partition is deadlocked and it is recommended that the partition be immediately terminated."

Not followed.

7 Implementation Defined Characteristics

In addition to the implementation dependent pragmas and attributes, and the implementation advice, there are a number of other Ada features that are potentially implementation dependent and are designated as implementation-defined. These are mentioned throughout the Ada Reference Manual, and are summarized in Annex M.

A requirement for conforming Ada compilers is that they provide documentation describing how the implementation deals with each of these issues. In this chapter you will find each point in Annex M listed, followed by a description of how GNAT handles the implementation dependence.

You can use this chapter as a guide to minimizing implementation dependent features in your programs if portability to other compilers and other operating systems is an important consideration. The numbers in each entry below correspond to the paragraph numbers in the Ada Reference Manual.

* "Whether or not each recommendation given in Implementation Advice is followed. See 1.1.2(37)."

See [Implementation Advice], page 159.

* "Capacity limitations of the implementation. See 1.1.3(3)."

The complexity of programs that can be processed is limited only by the total amount of available virtual memory, and disk space for the generated object files.

* "Variations from the standard that are impractical to avoid given the implementation's execution environment. See 1.1.3(6)."

There are no variations from the standard.

* "Which code_statements cause external interactions. See 1.1.3(10)."

Any 'code_statement' can potentially cause external interactions.

* "The coded representation for the text of an Ada program. See 2.1(4)."

See separate section on source representation.

* "The semantics of an Ada program whose text is not in Normalization Form C. See 2.1(4)."

See separate section on source representation.

* "The representation for an end of line. See 2.2(2)."

See separate section on source representation.

* "Maximum supported line length and lexical element length. See 2.2(15)."

The maximum line length is 255 characters and the maximum length of a lexical element is also 255 characters. This is the default setting if not overridden by the use of compiler switch '-gnaty' (which sets the maximum to 79) or '-gnatyMnn' which allows the maximum line length to be specified to be any value up to 32767. The maximum length of a lexical element is the same as the maximum line length.

* "Implementation defined pragmas. See 2.8(14)."

See [Implementation Defined Pragmas], page 4.

* "Effect of pragma `Optimize`. See 2.8(27)."

Pragma `Optimize`, if given with a `Time` or `Space` parameter, checks that the optimization flag is set, and aborts if it is not.

* "The message string associated with the Assertion_Error exception raised by the failure of a predicate check if there is no applicable Predicate_Failure aspect. See 3.2.4(31)."

In the case of a Dynamic_Predicate aspect, the string is "Dynamic_Predicate failed at <source position>", where "<source position>" might be something like "foo.adb:123". The Static_Predicate case is handled analogously.

* "The predefined integer types declared in `Standard`. See 3.5.4(25)."

Type	**Representation**
'Short_Short_Integer'	8-bit signed
'Short_Integer'	16-bit signed
'Integer'	32-bit signed
'Long_Integer'	64-bit signed (on most 64-bit targets, depending on the C definition of long) 32-bit signed (on all other targets)
'Long_Long_Integer'	64-bit signed
'Long_Long_Long_Integer'	128-bit signed (on 64-bit targets) 64-bit signed (on 32-bit targets)

* "Any nonstandard integer types and the operators defined for them. See 3.5.4(26)."

There are no nonstandard integer types.

* "Any nonstandard real types and the operators defined for them. See 3.5.6(8)."

There are no nonstandard real types.

* "What combinations of requested decimal precision and range are supported for floating point types. See 3.5.7(7)."

The precision and range are defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019).

* "The predefined floating point types declared in `Standard`. See 3.5.7(16)."

Type	**Representation**
'Short_Float'	IEEE Binary32 (Single)
'Float'	IEEE Binary32 (Single)
'Long_Float'	IEEE Binary64 (Double)

'Long_Long_Float' IEEE Binary64 (Double) on non-x86 architectures IEEE 80-bit Extended on x86 architecture

The default rounding mode specified by the IEEE 754 Standard is assumed both for static and dynamic computations (that is, round to nearest, ties to even). The input routines yield correctly rounded values for Short_Float, Float, and Long_Float at least. The output routines can compute up to twice as many exact digits as the value of `T'Digits` for any type, for example 30 digits for Long_Float; if more digits are requested, zeros are printed.

* "The small of an ordinary fixed point type. See 3.5.9(8)."

The small is the largest power of two that does not exceed the delta.

* "What combinations of small, range, and digits are supported for fixed point types. See 3.5.9(10)."

For an ordinary fixed point type, on 32-bit platforms, the small must lie in 2.0**(-80) .. 2.0**80 and the range in -9.0E+36 .. 9.0E+36; any combination is permitted that does not result in a mantissa larger than 63 bits.

On 64-bit platforms, the small must lie in 2.0**(-127) .. 2.0**127 and the range in -1.0E+76 .. 1.0E+76; any combination is permitted that does not result in a mantissa larger than 63 bits, and any combination is permitted that results in a mantissa between 64 and 127 bits if the small is the ratio of two integers that lie in 1 .. 2.0**127.

If the small is the ratio of two integers with 64-bit magnitude on 32-bit platforms and 128-bit magnitude on 64-bit platforms, which is the case if no `small` clause is provided, then the operations of the fixed point type are entirely implemented by means of integer instructions. In the other cases, some operations, in particular input and output, may be implemented by means of floating-point instructions and may be affected by accuracy issues on architectures other than x86.

For a decimal fixed point type, on 32-bit platforms, the small must lie in 1.0E-18 .. 1.0E+18 and the digits in 1 .. 18. On 64-bit platforms, the small must lie in 1.0E-38 .. 1.0E+38 and the digits in 1 .. 38.

* "The result of `Tags.Expanded_Name` for types declared within an unnamed 'block_statement'. See 3.9(10)."

Block numbers of the form `B`*`nnn`*, where 'nnn' is a decimal integer are allocated.

* "The sequence of characters of the value returned by Tags.Expanded_Name (respectively, Tags.Wide_Expanded_Name) when some of the graphic characters of Tags.Wide_Wide_Expanded_Name are not defined in Character (respectively, Wide_Character). See 3.9(10.1)."

This is handled in the same way as the implementation-defined behavior referenced in A.4.12(34).

* "Implementation-defined attributes. See 4.1.4(12)."

See [Implementation Defined Attributes], page 120.

* "The value of the parameter to Empty for some container aggregates. See 4.3.5(40)."

As per the suggestion given in the Annotated Ada RM, the default value of the formal parameter is used if one exists and zero is used otherwise.

* “The maximum number of chunks for a parallel reduction expression without a chunk_specification. See 4.5.10(21).”

Feature unimplemented.

* “Rounding of real static expressions which are exactly half-way between two machine numbers. See 4.9(38).”

Round to even is used in all such cases.

* “The maximum number of chunks for a parallel generalized iterator without a chunk_specification. See 5.5.2(10).”

Feature unimplemented.

* “The number of chunks for an array component iterator. See 5.5.2(11).”

Feature unimplemented.

* “Any extensions of the Global aspect. See 6.1.2(43).”

Feature unimplemented.

* “The circumstances the implementation passes in the null value for a view conversion of an access type used as an out parameter. See 6.4.1(19).”

Difficult to characterize.

* “Any extensions of the Default_Initial_Condition aspect. See 7.3.3(11).”

SPARK allows specifying ‘null’ as the Default_Initial_Condition aspect of a type. See the SPARK reference manual for further details.

* “Any implementation-defined time types. See 9.6(6).”

There are no implementation-defined time types.

* “The time base associated with relative delays. See 9.6(20).”

See 9.6(20). The time base used is that provided by the C library function `gettimeofday`.

* “The time base of the type `Calendar.Time`. See 9.6(23).”

The time base used is that provided by the C library function `gettimeofday`.

* “The time zone used for package `Calendar` operations. See 9.6(24).”

The time zone used by package `Calendar` is the current system time zone setting for local time, as accessed by the C library function `localtime`.

* “Any limit on ‘delay_until_statements’ of ‘select_statements’. See 9.6(29).”

There are no such limits.

* “The result of Calendar.Formatting.Image if its argument represents more than 100 hours. See 9.6.1(86).”

Calendar.Time_Error is raised.

* “Implementation-defined conflict check policies. See 9.10.1(5).”

There are no implementation-defined conflict check policies.

* “The representation for a compilation. See 10.1(2).”

A compilation is represented by a sequence of files presented to the compiler in a single invocation of the 'gcc' command.

* "Any restrictions on compilations that contain multiple compilation_units. See 10.1(4)."

No single file can contain more than one compilation unit, but any sequence of files can be presented to the compiler as a single compilation.

* "The mechanisms for creating an environment and for adding and replacing compilation units. See 10.1.4(3)."

See separate section on compilation model.

* "The manner of explicitly assigning library units to a partition. See 10.2(2)."

If a unit contains an Ada main program, then the Ada units for the partition are determined by recursive application of the rules in the Ada Reference Manual section 10.2(2-6). In other words, the Ada units will be those that are needed by the main program, and then this definition of need is applied recursively to those units, and the partition contains the transitive closure determined by this relationship. In short, all the necessary units are included, with no need to explicitly specify the list. If additional units are required, e.g., by foreign language units, then all units must be mentioned in the context clause of one of the needed Ada units.

If the partition contains no main program, or if the main program is in a language other than Ada, then GNAT provides the binder options '-z' and '-n' respectively, and in this case a list of units can be explicitly supplied to the binder for inclusion in the partition (all units needed by these units will also be included automatically). For full details on the use of these options, refer to 'GNAT Make Program gnatmake' in the *GNAT User's Guide*.

* "The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. See 10.2(2)."

The units needed by a given compilation unit are as defined in the Ada Reference Manual section 10.2(2-6). There are no implementation-defined pragmas or other implementation-defined means for specifying needed units.

* "The manner of designating the main subprogram of a partition. See 10.2(7)."

The main program is designated by providing the name of the corresponding `ALI` file as the input parameter to the binder.

* "The order of elaboration of 'library_items'. See 10.2(18)."

The first constraint on ordering is that it meets the requirements of Chapter 10 of the Ada Reference Manual. This still leaves some implementation-dependent choices, which are resolved by analyzing the elaboration code of each unit and identifying implicit elaboration-order dependencies.

* "Parameter passing and function return for the main subprogram. See 10.2(21)."

The main program has no parameters. It may be a procedure, or a function returning an integer type. In the latter case, the returned integer value is the return code of the program (overriding any value that may have been set by a call to `Ada.Command_Line.Set_Exit_Status`).

* "The mechanisms for building and running partitions. See 10.2(24)."

GNAT itself supports programs with only a single partition. The PolyORB product (which also includes an implementation of the PCS) provides a completely flexible method for building and running programs consisting of multiple partitions. 'NB!' See the note in [PolyORB], page 372, regarding the lifetime of this product.

* "The details of program execution, including program termination. See 10.2(25)."

See separate section on compilation model.

* "The semantics of any non-active partitions supported by the implementation. See 10.2(28)."

Passive partitions are supported on targets where shared memory is provided by the operating system. 'NB!' See the note in [PolyORB], page 372, regarding the lifetime of this product.

* "The information returned by `Exception_Message`. See 11.4.1(10)."

Exception message returns the null string unless a specific message has been passed by the program.

* "The result of `Exceptions.Exception_Name` for types declared within an unnamed 'block_statement'. See 11.4.1(12)."

Blocks have implementation defined names of the form `Bnnn` where 'nnn' is an integer.

* "The information returned by `Exception_Information`. See 11.4.1(13)."

`Exception_Information` returns a string in the following format:

```
*Exception_Name:* nnnnn
*Message:* mmmmm
*PID:* ppp
*Load address:* 0xhhhh
*Call stack traceback locations:*
0xhhhh 0xhhhh 0xhhhh ... 0xhhh
```

where

  * `nnnn` is the fully qualified name of the exception in all upper case letters. This line is always present.
  * `mmmm` is the message (this line present only if message is non-null)
  * `ppp` is the Process Id value as a decimal integer (this line is present only if the Process Id is nonzero). Currently we are not making use of this field.
  * The Load address line, the Call stack traceback locations line and the following values are present only if at least one traceback location was recorded. The Load address indicates the address at which the main executable was loaded; this line may not be present if operating system hasn't relocated the main executable. The values are given in C style format, with lower case letters for a-f, and only as many digits present as are necessary. The line terminator sequence at the end of each line, including the last line is a single `LF` character (`16#0A#`).

* "The sequence of characters of the value returned by Exceptions.Exception_Name (respectively, Exceptions.Wide_Exception_Name) when some of the graphic characters

of Exceptions.Wide_Wide_Exception_Name are not defined in Character (respectively, Wide_Character). See 11.4.1(12.1)."

This is handled in the same way as the implementation-defined behavior referenced in A.4.12(34).

* "The information returned by Exception_Information. See 11.4.1(13)."

The exception name and the source location at which the exception was raised are included.

* "Implementation-defined policy_identifiers and assertion_aspect_marks allowed in a pragma Assertion_Policy. See 11.4.2(9)."

Implementation-defined assertion_aspect_marks include Assert_And_Cut, Assume, Contract_Cases, Debug, Ghost, Initial_Condition, Loop_Invariant, Loop_Variant, Postcondition, Precondition, Predicate, Refined_Post, Statement_Assertions, and Subprogram_Variant. Implementation-defined policy_identifiers include Disable and Suppressible.

* "The default assertion policy. See 11.4.2(10)."

The default assertion policy is Ignore, although this can be overridden via compiler switches such as "-gnata".

* "Implementation-defined check names. See 11.5(27)."

The implementation-defined check names include Alignment_Check, Container_Checks, Duplicated_Tag_Check, Predicate_Check, Raise_Check, Tampering_Check, and Validity_Check. In addition, a user program can add implementation-defined check names by means of the pragma Check_Name. See the description of pragma `Suppress` for details.

* "Existence and meaning of second parameter of pragma Unsuppress. See 11.5(27.1)."

The legality rules for and semantics of the second parameter of pragma Unsuppress match those for the second argument of pragma Suppress.

* "The cases that cause conflicts between the representation of the ancestors of a type_declaration. See 13.1(13.1)."

No such cases exist.

* "The interpretation of each representation aspect. See 13.1(20)."

See separate section on data representations.

* "Any restrictions placed upon the specification of representation aspects. See 13.1(20)."

See separate section on data representations.

* "Implementation-defined aspects, including the syntax for specifying such aspects and the legality rules for such aspects. See 13.1.1(38)."

See [Implementation Defined Aspects], page 108.

* "The set of machine scalars. See 13.3(8.1)."

See separate section on data representations.

* "The meaning of `Size` for indefinite subtypes. See 13.3(48)."

The Size attribute of an indefinite subtype is not less than the Size attribute of any object of that type.

* "The meaning of Object_Size for indefinite subtypes. See 13.3(58)."

The Object_Size attribute of an indefinite subtype is not less than the Object_Size attribute of any object of that type.

* "The default external representation for a type tag. See 13.3(75)."

The default external representation for a type tag is the fully expanded name of the type in upper case letters.

* "What determines whether a compilation unit is the same in two different partitions. See 13.3(76)."

A compilation unit is the same in two different partitions if and only if it derives from the same source file.

* "Implementation-defined components. See 13.5.1(15)."

The only implementation defined component is the tag for a tagged type, which contains a pointer to the dispatching table.

* "If `Word_Size` = `Storage_Unit`, the default bit ordering. See 13.5.3(5)."

`Word_Size` does not equal `Storage_Unit` in this implementation.

* "The contents of the visible part of package `System`. See 13.7(2)."

See the definition of package System in `system.ads`. Note that two declarations are added to package System.

```
Max_Priority           : constant Positive := Priority'Last;
Max_Interrupt_Priority : constant Positive := Interrupt_Priority'Last;
```

* "The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.Storage_Element, and the declaration of Storage_Elements.Integer_Address. See 13.7.1(11)."

See the definition of package System.Storage_Elements in `s-stoele.ads`.

* "The contents of the visible part of package `System.Machine_Code`, and the meaning of 'code_statements'. See 13.8(7)."

See the definition and documentation in file `s-maccod.ads`.

* "The result of unchecked conversion for instances with scalar result types whose result is not defined by the language. See 13.9(11)."

Unchecked conversion between types of the same size results in an uninterpreted transmission of the bits from one type to the other. If the types are of unequal sizes, then in the case of discrete types, a shorter source is first zero or sign extended as necessary, and a shorter target is simply truncated on the left. For all non-discrete types, the source is first copied if necessary to ensure that the alignment requirements of the target are met, then a pointer is constructed to the source value, and the result is obtained by dereferencing this pointer after converting it to be a pointer to the target type. Unchecked conversions where the target subtype is an unconstrained array are not permitted. If the target alignment is greater than the source alignment, then a copy of the result is made with appropriate alignment

* "The result of unchecked conversion for instances with nonscalar result types whose result is not defined by the language. See 13.9(11)."

See preceding definition for the scalar result case.

* "Whether or not the implementation provides user-accessible names for the standard pool type(s). See 13.11(17)."

There are 3 different standard pools used by the compiler when `Storage_Pool` is not specified depending whether the type is local to a subprogram or defined at the library level and whether `Storage_Size``is specified or not. See documentation in the runtime library units ``System.Pool_Global`, `System.Pool_Size` and `System.Pool_Local` in files `s-poosiz.ads`, `s-pooglo.ads` and `s-pooloc.ads` for full details on the default pools used. All these pools are accessible by means of *with*ing these units.

* "The meaning of `Storage_Size` when neither the Storage_Size nor the Storage_Pool is specified for an access type. See 13.11(18)."

`Storage_Size` is measured in storage units, and refers to the total space available for an access type collection, or to the primary stack space for a task.

* "The effect of specifying aspect Default_Storage_Pool on an instance of a language-defined generic unit. See 13.11.3(5)."

Instances of language-defined generic units are treated the same as other instances with respect to the Default_Storage_Pool aspect.

* "Implementation-defined restrictions allowed in a pragma `Restrictions`. See 13.12(8.7)."

See [Standard and Implementation Defined Restrictions], page 144.

* "The consequences of violating limitations on `Restrictions` pragmas. See 13.12(9)."

Restrictions that can be checked at compile time are enforced at compile time; violations are illegal. For other restrictions, any violation during program execution results in erroneous execution.

* "Implementation-defined usage profiles allowed in a pragma Profile. See 13.12(15)."

See [Implementation Defined Pragmas], page 4.

* "The contents of the stream elements read and written by the Read and Write attributes of elementary types. See 13.13.2(9)."

The representation is the in-memory representation of the base type of the type, using the number of bits corresponding to the `type'Size` value, and the natural ordering of the machine.

* "The names and characteristics of the numeric subtypes declared in the visible part of package `Standard`. See A.1(3)."

See items describing the integer and floating-point types supported.

* "The values returned by Strings.Hash. See A.4.9(3)."

This hash function has predictable collisions and is subject to equivalent substring attacks. It is not suitable for construction of a hash table keyed on possibly malicious user input.

* "The value returned by a call to a Text_Buffer Get procedure if any character in the returned sequence is not defined in Character. See A.4.12(34)."

The contents of a buffer is represented internally as a UTF_8 string. The value return by Text_Buffer.Get is the result of passing that UTF_8 string to UTF_Encoding.Strings.Decode.

* "The value returned by a call to a Text_Buffer Wide_Get procedure if any character in the returned sequence is not defined in Wide_Character. See A.4.12(34)."

The contents of a buffer is represented internally as a UTF_8 string. The value return by Text_Buffer.Wide_Get is the result of passing that UTF_8 string to UTF_Encoding.Wide_Strings.Decode.

* "The accuracy actually achieved by the elementary functions. See A.5.1(1)."

The elementary functions correspond to the functions available in the C library. Only fast math mode is implemented.

* "The sign of a zero result from some of the operators or functions in `Numerics.Generic_Elementary_Functions`, when `Float_Type'Signed_Zeros` is `True`. See A.5.1(46)."

The sign of zeroes follows the requirements of the IEEE 754 standard on floating-point.

* "The value of `Numerics.Float_Random.Max_Image_Width`. See A.5.2(27)."

Maximum image width is 6864, see library file `s-rannum.ads`.

* "The value of `Numerics.Discrete_Random.Max_Image_Width`. See A.5.2(27)."

Maximum image width is 6864, see library file `s-rannum.ads`.

* "The string representation of a random number generator's state. See A.5.2(38)."

The value returned by the Image function is the concatenation of the fixed-width decimal representations of the 624 32-bit integers of the state vector.

* "The values of the `Model_Mantissa`, `Model_Emin`, `Model_Epsilon`, `Model`, `Safe_First`, and `Safe_Last` attributes, if the Numerics Annex is not supported. See A.5.3(72)."

Running the compiler with '-gnatS' to produce a listing of package `Standard` displays the values of these attributes.

* "The value of `Buffer_Size` in `Storage_IO`. See A.9(10)."

All type representations are contiguous, and the `Buffer_Size` is the value of `type'Size` rounded up to the next storage unit boundary.

* "External files for standard input, standard output, and standard error See A.10(5)."

These files are mapped onto the files provided by the C streams libraries. See source file `i-cstrea.ads` for further details.

* "The accuracy of the value produced by `Put`. See A.10.9(36)."

If more digits are requested in the output than are represented by the precision of the value, zeroes are output in the corresponding least significant digit positions.

* "Current size for a stream file for which positioning is not supported. See A.12.1(1.1)."

Positioning is supported.

* "The meaning of `Argument_Count`, `Argument`, and `Command_Name`. See A.15(1)."

These are mapped onto the `argv` and `argc` parameters of the main program in the natural manner.

* "The interpretation of file names and directory names. See A.16(46)."

These names are interpreted consistently with the underlying file system.

* "The maxium value for a file size in Directories. See A.16(87)."

Directories.File_Size'Last is equal to Long_Long_Integer'Last .

* "The result for Directories.Size for a directory or special file. See A.16(93)."

Name_Error is raised.

* "The result for Directories.Modification_Time for a directory or special file. See A.16(93)."

Name_Error is raised.

* "The interpretation of a nonnull search pattern in Directories. See A.16(104)."

When the `Pattern` parameter is not the null string, it is interpreted according to the syntax of regular expressions as defined in the `GNAT.Regexp` package.

See [GNAT.Regexp (g-regexp.ads)], page 272.

* "The results of a Directories search if the contents of the directory are altered while a search is in progress. See A.16(110)."

The effect of a call to Get_Next_Entry is determined by the current state of the directory.

* "The definition and meaning of an environment variable. See A.17(1)."

This definition is determined by the underlying operating system.

* "The circumstances where an environment variable cannot be defined. See A.17(16)."

There are no such implementation-defined circumstances.

* "Environment names for which Set has the effect of Clear. See A.17(17)."

There are no such names.

* "The value of Containers.Hash_Type'Modulus. The value of Containers.Count_Type'Last. See A.18.1(7)."

Containers.Hash_Type'Modulus is 2**32. Containers.Count_Type'Last is 2**31 - 1.

* "Implementation-defined convention names. See B.1(11)."

The following convention names are supported

Convention Name	**Interpretation**
'Ada'	Ada
'Ada_Pass_By_Copy'	Allowed for any types except by-reference types such as limited records. Compatible with convention Ada, but causes any parameters with this convention to be passed by copy.
'Ada_Pass_By_Reference'	Allowed for any types except by-copy types such as scalars. Compatible with convention Ada, but causes any parameters with this convention to be passed by reference.

'Assembler'	Assembly language
'Asm'	Synonym for Assembler
'Assembly'	Synonym for Assembler
'C'	C
'C_Pass_By_Copy'	Allowed only for record types, like C, but also notes that record is to be passed by copy rather than reference.
'COBOL'	COBOL
'C_Plus_Plus (or CPP)'	C++
'Default'	Treated the same as C
'External'	Treated the same as C
'Fortran'	Fortran
'Intrinsic'	For support of pragma `Import` with convention Intrinsic, see separate section on Intrinsic Subprograms.
'Stdcall'	Stdcall (used for Windows implementations only). This convention correspond to the WINAPI (previously called Pascal convention) C/C++ convention under Windows. A routine with this convention cleans the stack before exit. This pragma cannot be applied to a dispatching call.
'DLL'	Synonym for Stdcall
'Win32'	Synonym for Stdcall
'Stubbed'	Stubbed is a special convention used to indicate that the body of the subprogram will be entirely ignored. Any call to the subprogram is converted into a raise of the `Program_Error` exception. If a pragma `Import` specifies convention `stubbed` then no body need be present at all. This convention is useful during development for the inclusion of subprograms whose body has not yet been written. In addition, all otherwise unrecognized convention names are also treated as being synonymous with convention C. In all implementations, use of such other names results in a warning.

* "The meaning of link names. See B.1(36)."

Link names are the actual names used by the linker.

* "The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified. See B.1(36)."

The default linker name is that which would be assigned by the relevant external language, interpreting the Ada name as being in all lower case letters.

* "The effect of pragma `Linker_Options`. See B.1(37)."

The string passed to `Linker_Options` is presented uninterpreted as an argument to the link command, unless it contains ASCII.NUL characters. NUL characters if they appear act as argument separators, so for example

```
pragma Linker_Options ("-labc" & ASCII.NUL & "-ldef");
```

causes two separate arguments `-labc` and `-ldef` to be passed to the linker. The order of linker options is preserved for a given unit. The final list of options passed to the linker is in reverse order of the elaboration order. For example, linker options for a body always appear before the options from the corresponding package spec.

* "The contents of the visible part of package `Interfaces` and its language-defined descendants. See B.2(1)."

See files with prefix `i-` in the distributed library.

* "Implementation-defined children of package `Interfaces`. The contents of the visible part of package `Interfaces`. See B.2(11)."

See files with prefix `i-` in the distributed library.

* "The definitions of certain types and constants in Interfaces.C. See B.3(41)."

See source file `i-c.ads`.

* "The types `Floating`, `Long_Floating`, `Binary`, `Long_Binary`, `Decimal_ Element`, and `COBOL_Character`; and the initialization of the variables `Ada_To_COBOL` and `COBOL_To_Ada`, in `Interfaces.COBOL`. See B.4(50)."

COBOL	**Ada**
'Floating'	Float
'Long_Floating'	(Floating) Long_Float
'Binary'	Integer
'Long_Binary'	Long_Long_Integer
'Decimal_Element'	Character
'COBOL_Character'	Character

For initialization, see the file `i-cobol.ads` in the distributed library.

* "The types Fortran_Integer, Real, Double_Precision, and Character_Set in Interfaces.Fortran. See B.5(17)."

See source file `i-fortra.ads`. These types are derived, respectively, from Integer, Float, Long_Float, and Character.

* "Implementation-defined intrinsic subprograms. See C.1(1)."

See separate section on Intrinsic Subprograms.

* "Any restrictions on a protected procedure or its containing type when an aspect Attach_handler or Interrupt_Handler is specified. See C.3.1(17)."

There are no such restrictions.

* "Any other forms of interrupt handler supported by the Attach_Handler and Interrupt_Handler aspects. See C.3.1(19)."

There are no such forms.

* "The semantics of some attributes and functions of an entity for which aspect Discard_Names is True. See C.5(7)."

If Discard_Names is True for an enumeration type, the Image attribute provides the image of the Pos of the literal, and Value accepts Pos values.

If both of the aspects‘‘Discard_Names‘‘ and `No_Tagged_Streams` are true for a tagged type, its Expanded_Name and External_Tag values are empty strings. This is useful to avoid exposing entity names at binary level.

* "The modulus and size of Test_and_Set_Flag. See C.6.3(8)."

The modulus is 2**8. The size is 8.

* "The value used to represent the set value for Atomic_Test_and_Set. See C.6.3(10)."

The value is 1.

* "The result of the `Task_Identification.Image` attribute. See C.7.1(7)."

The result of this attribute is a string that identifies the object or component that denotes a given task. If a variable `Var` has a task type, the image for this task will have the form `Var_`*`XXXXXXXX`*, where the suffix 'XXXXXXXX' is the hexadecimal representation of the virtual address of the corresponding task control block. If the variable is an array of tasks, the image of each task will have the form of an indexed component indicating the position of a given task in the array, e.g., `Group(5)_`*`XXXXXXX`*. If the task is a component of a record, the image of the task will have the form of a selected component. These rules are fully recursive, so that the image of a task that is a subcomponent of a composite object corresponds to the expression that designates this task.

If a task is created by an allocator, its image depends on the context. If the allocator is part of an object declaration, the rules described above are used to construct its image, and this image is not affected by subsequent assignments. If the allocator appears within an expression, the image includes only the name of the task type.

If the configuration pragma Discard_Names is present, or if the restriction No_Implicit_Heap_Allocation is in effect, the image reduces to the numeric suffix, that is to say the hexadecimal representation of the virtual address of the control block of the task.

* "The value of `Current_Task` when in a protected entry or interrupt handler. See C.7.1(17)."

Protected entries or interrupt handlers can be executed by any convenient thread, so the value of `Current_Task` is undefined.

* "Granularity of locking for Task_Attributes. See C.7.2(16)."

No locking is needed if the formal type Attribute has the size and alignment of either Integer or System.Address and the bit representation of Initial_Value is all zeroes. Otherwise, locking is performed.

* "The declarations of `Any_Priority` and `Priority`. See D.1(11)."

See declarations in file `system.ads`.

* "Implementation-defined execution resources. See D.1(15)."

There are no implementation-defined execution resources.

* "Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. See D.2.1(3)."

On a multi-processor, a task that is waiting for access to a protected object does not keep its processor busy.

* "The affect of implementation defined execution resources on task dispatching. See D.2.1(9)."

Tasks map to threads in the threads package used by GNAT. Where possible and appropriate, these threads correspond to native threads of the underlying operating system.

* "Implementation-defined task dispatching policies. See D.2.2(3)."

There are no implementation-defined task dispatching policies.

* "The value of Default_Quantum in Dispatching.Round_Robin. See D.2.5(4)."

The value is 10 milliseconds.

* "Implementation-defined 'policy_identifiers' allowed in a pragma `Locking_Policy`. See D.3(4)."

The two implementation defined policies permitted in GNAT are `Inheritance_Locking` and `Concurrent_Readers_Locking`. On targets that support the `Inheritance_Locking` policy, locking is implemented by inheritance, i.e., the task owning the lock operates at a priority equal to the highest priority of any task currently requesting the lock. On targets that support the `Concurrent_Readers_Locking` policy, locking is implemented with a read/write lock allowing multiple protected object functions to enter concurrently.

* "Default ceiling priorities. See D.3(10)."

The ceiling priority of protected objects of the type `System.Interrupt_Priority'Last` as described in the Ada Reference Manual D.3(10),

* "The ceiling of any protected object used internally by the implementation. See D.3(16)."

The ceiling priority of internal protected objects is `System.Priority'Last`.

* "Implementation-defined queuing policies. See D.4(1)."

There are no implementation-defined queuing policies.

* "Implementation-defined admission policies. See D.4.1(1)."

There are no implementation-defined admission policies.

* "Any operations that implicitly require heap storage allocation. See D.7(8)."

The only operation that implicitly requires heap storage allocation is task creation.

* "When restriction No_Dynamic_CPU_Assignment applies to a partition, the processor on which a task with a CPU value of a Not_A_Specific_CPU will execute. See D.7(10)."

Unknown.

* "When restriction No_Task_Termination applies to a partition, what happens when a task terminates. See D.7(15.1)."

Execution is erroneous in that case.

* "The behavior when restriction Max_Storage_At_Blocking is violated. See D.7(17)."

Execution is erroneous in that case.

* "The behavior when restriction Max_Asynchronous_Select_Nesting is violated. See D.7(18)."

Execution is erroneous in that case.

* "The behavior when restriction Max_Tasks is violated. See D.7(19)."

Execution is erroneous in that case.

* "Whether the use of pragma Restrictions results in a reduction in program code or data size or execution time. See D.7(20)."

Yes it can, but the precise circumstances and properties of such reductions are difficult to characterize.

* "The value of Barrier_Limit'Last in Synchronous_Barriers. See D.10.1(4)."

Synchronous_Barriers.Barrier_Limit'Last is Integer'Last .

* "When an aborted task that is waiting on a Synchronous_Barrier is aborted. See D.10.1(13)."

Difficult to characterize.

* "The value of Min_Handler_Ceiling in Execution_Time.Group_Budgets. See D.14.2(7)."

See source file `a-etgrbu.ads`.

* "The value of CPU_Range'Last in System.Multiprocessors. See D.16(4)."

See source file `s-multip.ads`.

* "The processor on which the environment task executes in the absence of a value for the aspect CPU. See D.16(13)."

Unknown.

* "The means for creating and executing distributed programs. See E(5)."

The PolyORB product provides means creating and executing distributed programs. 'NB!' See the note in [PolyORB], page 372, regarding the lifetime of this product.

* "Any events that can result in a partition becoming inaccessible. See E.1(7)."

See the PolyORB user guide for full details on such events. 'NB!' Consider the note in [PolyORB], page 372, regarding the lifetime of this product.

* "The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases. See E.1(11)."

See the PolyORB user guide for full details on these aspects of multi-partition execution. 'NB!' Consider the note in [PolyORB], page 372, regarding the lifetime of this product.

* "Whether the execution of the remote subprogram is immediately aborted as a result of cancellation. See E.4(13)."

See the PolyORB user guide for details on the effect of abort in a distributed application. 'NB!' Consider the note in [PolyORB], page 372, regarding the lifetime of this product.

* "The range of type System.RPC.Partition_Id. See E.5(14)."

System.RPC.Partition_ID'Last is Integer'Last. See source file `s-rpc.ads`.

* "Implementation-defined interfaces in the PCS. See E.5(26)."

See the PolyORB user guide for a full description of all implementation defined interfaces. 'NB!' See the note in [PolyORB], page 372, regarding the lifetime of this product.

* "The values of named numbers in the package `Decimal`. See F.2(7)."

Named Number	**Value**
'Max_Scale'	+18
'Min_Scale'	-18
'Min_Delta'	1.0E-18
'Max_Delta'	1.0E+18
'Max_Decimal_Digits'	18

* "The value of `Max_Picture_Length` in the package `Text_IO.Editing`. See F.3.3(16)."

  64

* "The value of `Max_Picture_Length` in the package `Wide_Text_IO.Editing`. See F.3.4(5)."

  64

* "The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. See G.1(1)."

Standard library functions are used for the complex arithmetic operations. Only fast math mode is currently supported.

* "The sign of a zero result (or a component thereof) from any operator or function in `Numerics.Generic_Complex_Types`, when `Real'Signed_Zeros` is True. See G.1.1(53)."

The signs of zero values are as recommended by the relevant implementation advice.

* "The sign of a zero result (or a component thereof) from any operator or function in `Numerics.Generic_Complex_Elementary_Functions`, when `Real'Signed_Zeros` is `True`. See G.1.2(45)."

The signs of zero values are as recommended by the relevant implementation advice.

* "Whether the strict mode or the relaxed mode is the default. See G.2(2)."

The strict mode is the default. There is no separate relaxed mode. GNAT provides a highly efficient implementation of strict mode.

* "The result interval in certain cases of fixed-to-float conversion. See G.2.1(10)."

For cases where the result interval is implementation dependent, the accuracy is that provided by performing all operations in 64-bit IEEE floating-point format.

* "The result of a floating point arithmetic operation in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See G.2.1(13)."

Infinite and NaN values are produced as dictated by the IEEE floating-point standard. Note that on machines that are not fully compliant with the IEEE floating-point standard, such as Alpha, the '-mieee' compiler flag must be used for achieving IEEE conforming behavior (although at the cost of a significant performance penalty), so infinite and NaN values are properly generated.

* "The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal. See G.2.1(16)."

Not relevant, division is IEEE exact.

* "The definition of close result set, which determines the accuracy of certain fixed point multiplications and divisions. See G.2.3(5)."

Operations in the close result set are performed using IEEE long format floating-point arithmetic. The input operands are converted to floating-point, the operation is done in floating-point, and the result is converted to the target type.

* "Conditions on a 'universal_real' operand of a fixed point multiplication or division for which the result shall be in the perfect result set. See G.2.3(22)."

The result is only defined to be in the perfect result set if the result can be computed by a single scaling operation involving a scale factor representable in 64 bits.

* "The result of a fixed point arithmetic operation in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See G.2.3(27)."

Not relevant, `Machine_Overflows` is `True` for fixed-point types.

* "The result of an elementary function reference in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See G.2.4(4)."

IEEE infinite and Nan values are produced as appropriate.

* "The value of the angle threshold, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. See G.2.4(10)."

Information on this subject is not yet available.

* "The accuracy of certain elementary functions for parameters beyond the angle threshold. See G.2.4(10)."

Information on this subject is not yet available.

* “The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the `Machine_Overflows` attribute of the corresponding real type is `False`. See G.2.6(5).”

IEEE infinite and Nan values are produced as appropriate.

* “The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold. See G.2.6(8).”

Information on those subjects is not yet available.

* “The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix. See G.3.1(81).”

Information on those subjects is not yet available.

* “The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Complex_Matrix. See G.3.2(149).”

Information on those subjects is not yet available.

* “The consequences of violating No_Hidden_Indirect_Globals. See H.4(23.9).”

Execution is erroneous in that case.

8 Intrinsic Subprograms

GNAT allows a user application program to write the declaration:

```
pragma Import (Intrinsic, name);
```

providing that the name corresponds to one of the implemented intrinsic subprograms in GNAT, and that the parameter profile of the referenced subprogram meets the requirements. This chapter describes the set of implemented intrinsic subprograms, and the requirements on parameter profiles. Note that no body is supplied; as with other uses of pragma Import, the body is supplied elsewhere (in this case by the compiler itself). Note that any use of this feature is potentially non-portable, since the Ada standard does not require Ada compilers to implement this feature.

8.1 Intrinsic Operators

All the predefined numeric operators in package Standard in `pragma Import (Intrinsic,..)` declarations. In the binary operator case, the operands must have the same size. The operand or operands must also be appropriate for the operator. For example, for addition, the operands must both be floating-point or both be fixed-point, and the right operand for `"**"` must have a root type of `Standard.Integer'Base`. You can use an intrinsic operator declaration as in the following example:

```
type Int1 is new Integer;
type Int2 is new Integer;

function "+" (X1 : Int1; X2 : Int2) return Int1;
function "+" (X1 : Int1; X2 : Int2) return Int2;
pragma Import (Intrinsic, "+");
```

This declaration would permit 'mixed mode' arithmetic on items of the differing types `Int1` and `Int2`. It is also possible to specify such operators for private types, if the full views are appropriate arithmetic types.

8.2 Compilation_ISO_Date

This intrinsic subprogram is used in the implementation of the library package `GNAT.Source_Info`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Source_Info.Compilation_ISO_Date` to obtain the date of the current compilation (in local time format YYYY-MM-DD).

8.3 Compilation_Date

Same as Compilation_ISO_Date, except the string is in the form MMM DD YYYY.

8.4 Compilation_Time

This intrinsic subprogram is used in the implementation of the library package `GNAT.Source_Info`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Source_Info.Compilation_Time` to obtain the time of the current compilation (in local time format HH:MM:SS).

8.5 Enclosing_Entity

This intrinsic subprogram is used in the implementation of the library package `GNAT.Source_Info`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Source_Info.Enclosing_Entity` to obtain the name of the current subprogram, package, task, entry, or protected subprogram.

8.6 Exception_Information

This intrinsic subprogram is used in the implementation of the library package `GNAT.Current_Exception`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Current_Exception.Exception_Information` to obtain the exception information associated with the current exception.

8.7 Exception_Message

This intrinsic subprogram is used in the implementation of the library package `GNAT.Current_Exception`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Current_Exception.Exception_Message` to obtain the message associated with the current exception.

8.8 Exception_Name

This intrinsic subprogram is used in the implementation of the library package `GNAT.Current_Exception`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Current_Exception.Exception_Name` to obtain the name of the current exception.

8.9 File

This intrinsic subprogram is used in the implementation of the library package `GNAT.Source_Info`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Source_Info.File` to obtain the name of the current file.

8.10 Line

This intrinsic subprogram is used in the implementation of the library package `GNAT.Source_Info`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Source_Info.Line` to obtain the number of the current source line.

8.11 Shifts and Rotates

In standard Ada, the shift and rotate functions are available only for the predefined modular types in package `Interfaces`. However, in GNAT it is possible to define these functions for any integer type (signed or modular), as in this example:

```
function Shift_Left
  (Value  : T;
   Amount : Natural) return T
with Import, Convention => Intrinsic;
```

The function name must be one of Shift_Left, Shift_Right, Shift_Right_Arithmetic, Rotate_Left, or Rotate_Right. T must be an integer type. T'Size must be 8, 16, 32 or 64 bits; if T is modular, the modulus must be 2**8, 2**16, 2**32 or 2**64. The result type must be the same as the type of `Value`. The shift amount must be Natural. The formal parameter names can be anything.

A more convenient way of providing these shift operators is to use the Provide_Shift_Operators pragma, which provides the function declarations and corresponding pragma Import's for all five shift functions. For signed types the semantics of these operators is to interpret the bitwise result of the corresponding operator for modular type. In particular, shifting a negative number may change its sign bit to positive.

8.12 Source_Location

This intrinsic subprogram is used in the implementation of the library routine `GNAT.Source_Info`. The only useful use of the intrinsic import in this case is the one in this unit, so an application program should simply call the function `GNAT.Source_Info.Source_Location` to obtain the current source file location.

9 Representation Clauses and Pragmas

This section describes the representation clauses accepted by GNAT, and their effect on the representation of corresponding data objects.

GNAT fully implements Annex C (Systems Programming). This means that all the implementation advice sections in chapter 13 are fully implemented. However, these sections only require a minimal level of support for representation clauses. GNAT provides much more extensive capabilities, and this section describes the additional capabilities provided.

9.1 Alignment Clauses

GNAT requires that all alignment clauses specify 0 or a power of 2, and all default alignments are always a power of 2. Specifying 0 is the same as specifying 1.

The default alignment values are as follows:

* ‘Elementary Types’.

  For elementary types, the alignment is the minimum of the actual size of objects of the type divided by `Storage_Unit`, and the maximum default alignment supported by the target. (This maximum default alignment is given by the GNAT-specific attribute `Standard'Maximum_Alignment`; see [Attribute Maximum_Alignment], page 129.)

  For example, for type `Long_Float`, the object size is 8 bytes, and the default alignment will be 8 on any target that supports alignments this large, but on some targets, the maximum alignment may be smaller than 8, in which case objects of type `Long_Float` will be maximally aligned.

* ‘Arrays’.

  For arrays, the alignment is equal to the alignment of the component type for the normal case where no packing or component size is given. If the array is packed, and the packing is effective (see separate section on packed arrays), then the alignment will be either 4, 2, or 1 for long packed arrays or arrays whose length is not known at compile time, depending on whether the component size is divisible by 4, 2, or is odd. For short packed arrays, which are handled internally as modular types, the alignment will be as described for elementary types, e.g. a packed array of length 31 bits will have an object size of four bytes, and an alignment of 4.

* ‘Records’.

  For the normal unpacked case, the alignment of a record is equal to the maximum alignment of any of its components. For tagged records, this includes the implicit access type used for the tag. If a pragma `Pack` is used and all components are packable (see separate section on pragma `Pack`), then the resulting alignment is 1, unless the layout of the record makes it profitable to increase it.

  A special case is when:

  * the size of the record is given explicitly, or a full record representation clause is given, and
  * the size of the record is 2, 4, or 8 bytes.

  In this case, an alignment is chosen to match the size of the record. For example, if we have:

  ```
  type Small is record
  ```

```
      A, B : Character;
   end record;
   for Small'Size use 16;
```

then the default alignment of the record type `Small` is 2, not 1. This leads to more efficient code when the record is treated as a unit, and also allows the type to specified as `Atomic` on architectures requiring strict alignment.

An alignment clause may specify a larger alignment than the default value up to some maximum value dependent on the target. It may also specify a smaller alignment than the default value for enumeration, integer and fixed point types, as well as for record types, for example

```
type V is record
   A : Integer;
end record;

for V'alignment use 1;
```

The default alignment for the type `V` is 4, as a result of the Integer field in the record, but it is permissible, as shown, to override the default alignment of the record with a smaller value.

Note that according to the Ada standard, an alignment clause applies only to the first named subtype. If additional subtypes are declared, then the compiler is allowed to choose any alignment it likes, and there is no way to control this choice. Consider:

```
type R is range 1 .. 10_000;
for R'Alignment use 1;
subtype RS is R range 1 .. 1000;
```

The alignment clause specifies an alignment of 1 for the first named subtype `R` but this does not necessarily apply to `RS`. When writing portable Ada code, you should avoid writing code that explicitly or implicitly relies on the alignment of such subtypes.

For the GNAT compiler, if an explicit alignment clause is given, this value is also used for any subsequent subtypes. So for GNAT, in the above example, you can count on the alignment of `RS` being 1. But this assumption is non-portable, and other compilers may choose different alignments for the subtype `RS`.

9.2 Size Clauses

The default size for a type `T` is obtainable through the language-defined attribute `T'Size` and also through the equivalent GNAT-defined attribute `T'Value_Size`. For objects of type `T`, GNAT will generally increase the type size so that the object size (obtainable through the GNAT-defined attribute `T'Object_Size`) is a multiple of `T'Alignment * Storage_Unit`.

For example:

```
type Smallint is range 1 .. 6;

type Rec is record
   Y1 : integer;
   Y2 : boolean;
end record;
```

In this example, `Smallint'Size` = `Smallint'Value_Size` = 3, as specified by the RM rules, but objects of this type will have a size of 8 (`Smallint'Object_Size` = 8), since objects by default occupy an integral number of storage units. On some targets, notably older versions of the Digital Alpha, the size of stand alone objects of this type may be 32, reflecting the inability of the hardware to do byte load/stores.

Similarly, the size of type `Rec` is 40 bits (`Rec'Size` = `Rec'Value_Size` = 40), but the alignment is 4, so objects of this type will have their size increased to 64 bits so that it is a multiple of the alignment (in bits). This decision is in accordance with the specific Implementation Advice in RM 13.3(43):

> "A `Size` clause should be supported for an object if the specified `Size` is at least as large as its subtype's `Size`, and corresponds to a size in storage elements that is a multiple of the object's `Alignment` (if the `Alignment` is nonzero)."

An explicit size clause may be used to override the default size by increasing it. For example, if we have:

```
type My_Boolean is new Boolean;
for My_Boolean'Size use 32;
```

then values of this type will always be 32-bit long. In the case of discrete types, the size can be increased up to 64 bits on 32-bit targets and 128 bits on 64-bit targets, with the effect that the entire specified field is used to hold the value, sign- or zero-extended as appropriate. If more than 64 bits or 128 bits resp. is specified, then padding space is allocated after the value, and a warning is issued that there are unused bits.

Similarly the size of records and arrays may be increased, and the effect is to add padding bits after the value. This also causes a warning message to be generated.

The largest Size value permitted in GNAT is 2**31-1. Since this is a Size in bits, this corresponds to an object of size 256 megabytes (minus one). This limitation is true on all targets. The reason for this limitation is that it improves the quality of the code in many cases if it is known that a Size value can be accommodated in an object of type Integer.

9.3 Storage_Size Clauses

For tasks, the `Storage_Size` clause specifies the amount of space to be allocated for the task stack. This cannot be extended, and if the stack is exhausted, then `Storage_Error` will be raised (if stack checking is enabled). Use a `Storage_Size` attribute definition clause, or a `Storage_Size` pragma in the task definition to set the appropriate required size. A useful technique is to include in every task definition a pragma of the form:

```
pragma Storage_Size (Default_Stack_Size);
```

Then `Default_Stack_Size` can be defined in a global package, and modified as required. Any tasks requiring stack sizes different from the default can have an appropriate alternative reference in the pragma.

You can also use the '-d' binder switch to modify the default stack size.

For access types, the `Storage_Size` clause specifies the maximum space available for allocation of objects of the type. If this space is exceeded then `Storage_Error` will be raised by an allocation attempt. In the case where the access type is declared local to a subprogram, the use of a `Storage_Size` clause triggers automatic use of a special predefined storage pool

(`System.Pool_Size`) that ensures that all space for the pool is automatically reclaimed on exit from the scope in which the type is declared.

A special case recognized by the compiler is the specification of a `Storage_Size` of zero for an access type. This means that no items can be allocated from the pool, and this is recognized at compile time, and all the overhead normally associated with maintaining a fixed size storage pool is eliminated. Consider the following example:

```
procedure p is
   type R is array (Natural) of Character;
   type P is access all R;
   for P'Storage_Size use 0;
   --  Above access type intended only for interfacing purposes

   y : P;

   procedure g (m : P);
   pragma Import (C, g);

   --  ...

begin
   --  ...
   y := new R;
end;
```

As indicated in this example, these dummy storage pools are often useful in connection with interfacing where no object will ever be allocated. If you compile the above example, you get the warning:

```
p.adb:16:09: warning: allocation from empty storage pool
p.adb:16:09: warning: Storage_Error will be raised at run time
```

Of course in practice, there will not be any explicit allocators in the case of such an access declaration.

9.4 Size of Variant Record Objects

In the case of variant record objects, there is a question whether Size gives information about a particular variant, or the maximum size required for any variant. Consider the following program

```
with Text_IO; use Text_IO;
procedure q is
   type R1 (A : Boolean := False) is record
     case A is
       when True  => X : Character;
       when False => null;
     end case;
   end record;

   V1 : R1 (False);
```

```
   V2 : R1;

begin
   Put_Line (Integer'Image (V1'Size));
   Put_Line (Integer'Image (V2'Size));
end q;
```

Here we are dealing with a variant record, where the True variant requires 16 bits, and the False variant requires 8 bits. In the above example, both V1 and V2 contain the False variant, which is only 8 bits long. However, the result of running the program is:

```
8
16
```

The reason for the difference here is that the discriminant value of V1 is fixed, and will always be False. It is not possible to assign a True variant value to V1, therefore 8 bits is sufficient. On the other hand, in the case of V2, the initial discriminant value is False (from the default), but it is possible to assign a True variant value to V2, therefore 16 bits must be allocated for V2 in the general case, even fewer bits may be needed at any particular point during the program execution.

As can be seen from the output of this program, the `'Size` attribute applied to such an object in GNAT gives the actual allocated size of the variable, which is the largest size of any of the variants. The Ada Reference Manual is not completely clear on what choice should be made here, but the GNAT behavior seems most consistent with the language in the RM.

In some cases, it may be desirable to obtain the size of the current variant, rather than the size of the largest variant. This can be achieved in GNAT by making use of the fact that in the case of a subprogram parameter, GNAT does indeed return the size of the current variant (because a subprogram has no way of knowing how much space is actually allocated for the actual).

Consider the following modified version of the above program:

```
with Text_IO; use Text_IO;
procedure q is
   type R1 (A : Boolean := False) is record
     case A is
       when True  => X : Character;
       when False => null;
     end case;
   end record;

   V2 : R1;

   function Size (V : R1) return Integer is
   begin
      return V'Size;
   end Size;

begin
```

```
   Put_Line (Integer'Image (V2'Size));
   Put_Line (Integer'Image (Size (V2)));
   V2 := (True, 'x');
   Put_Line (Integer'Image (V2'Size));
   Put_Line (Integer'Image (Size (V2)));
end q;
```

The output from this program is

```
16
8
16
16
```

Here we see that while the `'Size` attribute always returns the maximum size, regardless of the current variant value, the `Size` function does indeed return the size of the current variant value.

9.5 Biased Representation

In the case of scalars with a range starting at other than zero, it is possible in some cases to specify a size smaller than the default minimum value, and in such cases, GNAT uses an unsigned biased representation, in which zero is used to represent the lower bound, and successive values represent successive values of the type.

For example, suppose we have the declaration:

```
type Small is range -7 .. -4;
for Small'Size use 2;
```

Although the default size of type `Small` is 4, the `Size` clause is accepted by GNAT and results in the following representation scheme:

```
-7 is represented as 2#00#
-6 is represented as 2#01#
-5 is represented as 2#10#
-4 is represented as 2#11#
```

Biased representation is only used if the specified `Size` clause cannot be accepted in any other manner. These reduced sizes that force biased representation can be used for all discrete types except for enumeration types for which a representation clause is given.

9.6 Value_Size and Object_Size Clauses

In Ada 95 and Ada 2005, `T'Size` for a type `T` is the minimum number of bits required to hold values of type `T`. Although this interpretation was allowed in Ada 83, it was not required, and this requirement in practice can cause some significant difficulties. For example, in most Ada 83 compilers, `Natural'Size` was 32. However, in Ada 95 and Ada 2005, `Natural'Size` is typically 31. This means that code may change in behavior when moving from Ada 83 to Ada 95 or Ada 2005. For example, consider:

```
type Rec is record
   A : Natural;
   B : Natural;
end record;
```

```
for Rec use record
   A at 0 range 0 .. Natural'Size - 1;
   B at 0 range Natural'Size .. 2 * Natural'Size - 1;
end record;
```

In the above code, since the typical size of `Natural` objects is 32 bits and `Natural'Size` is 31, the above code can cause unexpected inefficient packing in Ada 95 and Ada 2005, and in general there are cases where the fact that the object size can exceed the size of the type causes surprises.

To help get around this problem GNAT provides two implementation defined attributes, `Value_Size` and `Object_Size`. When applied to a type, these attributes yield the size of the type (corresponding to the RM defined size attribute), and the size of objects of the type respectively.

The `Object_Size` is used for determining the default size of objects and components. This size value can be referred to using the `Object_Size` attribute. The phrase 'is used' here means that it is the basis of the determination of the size. The backend is free to pad this up if necessary for efficiency, e.g., an 8-bit stand-alone character might be stored in 32 bits on a machine with no efficient byte access instructions such as the Alpha.

The default rules for the value of `Object_Size` for discrete types are as follows:

* The `Object_Size` for base subtypes reflect the natural hardware size in bits (run the compiler with '-gnatS' to find those values for numeric types). Enumeration types and fixed-point base subtypes have 8, 16, 32, or 64 bits for this size, depending on the range of values to be stored.
* The `Object_Size` of a subtype is the same as the `Object_Size` of the type from which it is obtained.
* The `Object_Size` of a derived base type is copied from the parent base type, and the `Object_Size` of a derived first subtype is copied from the parent first subtype.

The `Value_Size` attribute is the (minimum) number of bits required to store a value of the type. This value is used to determine how tightly to pack records or arrays with components of this type, and also affects the semantics of unchecked conversion (unchecked conversions where the `Value_Size` values differ generate a warning, and are potentially target dependent).

The default rules for the value of `Value_Size` are as follows:

* The `Value_Size` for a base subtype is the minimum number of bits required to store all values of the type (including the sign bit only if negative values are possible).
* If a subtype statically matches the first subtype of a given type, then it has by default the same `Value_Size` as the first subtype. (This is a consequence of RM 13.1(14): "if two subtypes statically match, then their subtype-specific aspects are the same".)
* All other subtypes have a `Value_Size` corresponding to the minimum number of bits required to store all values of the subtype. For dynamic bounds, it is assumed that the value can range down or up to the corresponding bound of the ancestor

The RM defined attribute `Size` corresponds to the `Value_Size` attribute.

The `Size` attribute may be defined for a first-named subtype. This sets the `Value_Size` of the first-named subtype to the given value, and the `Object_Size` of this first-named subtype to the given value padded up to an appropriate boundary. It is a consequence of the default rules above that this `Object_Size` will apply to all further subtypes. On the other hand, `Value_Size` is affected only for the first subtype, any dynamic subtypes obtained from it directly, and any statically matching subtypes. The `Value_Size` of any other static subtypes is not affected.

`Value_Size` and `Object_Size` may be explicitly set for any subtype using an attribute definition clause. Note that the use of these attributes can cause the RM 13.1(14) rule to be violated. If two access types reference aliased objects whose subtypes have differing `Object_Size` values as a result of explicit attribute definition clauses, then it is illegal to convert from one access subtype to the other. For a more complete description of this additional legality rule, see the description of the `Object_Size` attribute.

To get a feel for the difference, consider the following examples (note that in each case the base is `Short_Short_Integer` with a size of 8):

Type or subtype declaration	**Object_Size**	**Value_Size**
`type X1 is range 0 .. 5;`	8	3
`type X2 is range 0 .. 5; for X2'Size use 12;`	16	12
`subtype X3 is X2 range 0 .. 3;`	16	2
`subtype X4 is X2'Base range 0 .. 10;`	8	4
`Dynamic : X2'Base range -64 .. +63;`		
`subtype X5 is X2 range 0 .. Dynamic;`	16	3*
`subtype X6 is X2'Base range 0 .. Dynamic;`	8	7*

Note: the entries marked '*' are not actually specified by the Ada Reference Manual, which has nothing to say about size in the dynamic case. What GNAT does is to allocate sufficient bits to accommodate any possible dynamic values for the bounds at run-time.

So far, so good, but GNAT has to obey the RM rules, so the question is under what conditions must the RM `Size` be used. The following is a list of the occasions on which the RM `Size` must be used:

* Component size for packed arrays or records
* Value of the attribute `Size` for a type
* Warning about sizes not matching for unchecked conversion

For record types, the `Object_Size` is always a multiple of the alignment of the type (this is true for all types). In some cases the `Value_Size` can be smaller. Consider:

```
type R is record
  X : Integer;
  Y : Character;
end record;
```

On a typical 32-bit architecture, the X component will occupy four bytes and the Y component will occupy one byte, for a total of 5 bytes. As a result `R'Value_Size` will be 40 (bits) since this is the minimum size required to store a value of this type. For example, it is permissible to have a component of type R in an array whose component size is specified to be 40 bits.

However, `R'Object_Size` will be 64 (bits). The difference is due to the alignment requirement for objects of the record type. The X component will require four-byte alignment because that is what type Integer requires, whereas the Y component, a Character, will only require 1-byte alignment. Since the alignment required for X is the greatest of all the components' alignments, that is the alignment required for the enclosing record type, i.e., 4 bytes or 32 bits. As indicated above, the actual object size must be rounded up so that it is a multiple of the alignment value. Therefore, 40 bits rounded up to the next multiple of 32 yields 64 bits.

For all other types, the `Object_Size` and `Value_Size` are the same (and equivalent to the RM attribute `Size`). Only `Size` may be specified for such types.

Note that `Value_Size` can be used to force biased representation for a particular subtype. Consider this example:

```
type R is (A, B, C, D, E, F);
subtype RAB is R range A .. B;
subtype REF is R range E .. F;
```

By default, `RAB` has a size of 1 (sufficient to accommodate the representation of `A` and `B`, 0 and 1), and `REF` has a size of 3 (sufficient to accommodate the representation of `E` and `F`, 4 and 5). But if we add the following `Value_Size` attribute definition clause:

```
for REF'Value_Size use 1;
```

then biased representation is forced for `REF`, and 0 will represent `E` and 1 will represent `F`. A warning is issued when a `Value_Size` attribute definition clause forces biased representation. This warning can be turned off using `-gnatw.B`.

9.7 Component_Size Clauses

Normally, the value specified in a component size clause must be consistent with the subtype of the array component with regard to size and alignment. In other words, the value specified must be at least equal to the size of this subtype, and must be a multiple of the alignment value.

In addition, component size clauses are allowed which cause the array to be packed, by specifying a smaller value. A first case is for component size values in the range 1 through 63 on 32-bit targets, and 1 through 127 on 64-bit targets. The value specified may not be smaller than the Size of the subtype. GNAT will accurately honor all packing requests in this range. For example, if we have:

```
type r is array (1 .. 8) of Natural;
for r'Component_Size use 31;
```

then the resulting array has a length of 31 bytes (248 bits = 8 * 31). Of course access to the components of such an array is considerably less efficient than if the natural component size of 32 is used. A second case is when the subtype of the component is a record type padded because of its default alignment. For example, if we have:

```
type r is record
  i : Integer;
  j : Integer;
  b : Boolean;
end record;

type a is array (1 .. 8) of r;
for a'Component_Size use 72;
```

then the resulting array has a length of 72 bytes, instead of 96 bytes if the alignment of the record (4) was obeyed.

Note that there is no point in giving both a component size clause and a pragma Pack for the same array type. if such duplicate clauses are given, the pragma Pack will be ignored.

9.8 Bit_Order Clauses

For record subtypes, GNAT permits the specification of the `Bit_Order` attribute. The specification may either correspond to the default bit order for the target, in which case the specification has no effect and places no additional restrictions, or it may be for the non-standard setting (that is the opposite of the default).

In the case where the non-standard value is specified, the effect is to renumber bits within each byte, but the ordering of bytes is not affected. There are certain restrictions placed on component clauses as follows:

* Components fitting within a single storage unit.

  These are unrestricted, and the effect is merely to renumber bits. For example if we are on a little-endian machine with `Low_Order_First` being the default, then the following two declarations have exactly the same effect:

  ```
  type R1 is record
     A : Boolean;
     B : Integer range 1 .. 120;
  end record;

  for R1 use record
     A at 0 range 0 .. 0;
     B at 0 range 1 .. 7;
  end record;

  type R2 is record
     A : Boolean;
     B : Integer range 1 .. 120;
  end record;

  for R2'Bit_Order use High_Order_First;

  for R2 use record
     A at 0 range 7 .. 7;
     B at 0 range 0 .. 6;
  ```

```
end record;
```

The useful application here is to write the second declaration with the `Bit_Order` attribute definition clause, and know that it will be treated the same, regardless of whether the target is little-endian or big-endian.

* Components occupying an integral number of bytes.

  These are components that exactly fit in two or more bytes. Such component declarations are allowed, but have no effect, since it is important to realize that the `Bit_Order` specification does not affect the ordering of bytes. In particular, the following attempt at getting an endian-independent integer does not work:

```
type R2 is record
   A : Integer;
end record;

for R2'Bit_Order use High_Order_First;

for R2 use record
   A at 0 range 0 .. 31;
end record;
```

  This declaration will result in a little-endian integer on a little-endian machine, and a big-endian integer on a big-endian machine. If byte flipping is required for interoperability between big- and little-endian machines, this must be explicitly programmed. This capability is not provided by `Bit_Order`.

* Components that are positioned across byte boundaries.

  but do not occupy an integral number of bytes. Given that bytes are not reordered, such fields would occupy a non-contiguous sequence of bits in memory, requiring non-trivial code to reassemble. They are for this reason not permitted, and any component clause specifying such a layout will be flagged as illegal by GNAT.

Since the misconception that Bit_Order automatically deals with all endian-related incompatibilities is a common one, the specification of a component field that is an integral number of bytes will always generate a warning. This warning may be suppressed using `pragma Warnings (Off)` if desired. The following section contains additional details regarding the issue of byte ordering.

9.9 Effect of Bit_Order on Byte Ordering

In this section we will review the effect of the `Bit_Order` attribute definition clause on byte ordering. Briefly, it has no effect at all, but a detailed example will be helpful. Before giving this example, let us review the precise definition of the effect of defining `Bit_Order`. The effect of a non-standard bit order is described in section 13.5.3 of the Ada Reference Manual:

> "2 A bit ordering is a method of interpreting the meaning of the storage place attributes."

To understand the precise definition of storage place attributes in this context, we visit section 13.5.1 of the manual:

> "13 A record_representation_clause (without the mod_clause) specifies the layout. The storage place attributes (see 13.5.2) are taken from the values of the position, first_bit, and last_bit expressions after normalizing those values so that first_bit is less than Storage_Unit."

The critical point here is that storage places are taken from the values after normalization, not before. So the `Bit_Order` interpretation applies to normalized values. The interpretation is described in the later part of the 13.5.3 paragraph:

> "2 A bit ordering is a method of interpreting the meaning of the storage place attributes. High_Order_First (known in the vernacular as 'big endian') means that the first bit of a storage element (bit 0) is the most significant bit (interpreting the sequence of bits that represent a component as an unsigned integer value). Low_Order_First (known in the vernacular as 'little endian') means the opposite: the first bit is the least significant."

Note that the numbering is with respect to the bits of a storage unit. In other words, the specification affects only the numbering of bits within a single storage unit.

We can make the effect clearer by giving an example.

Suppose that we have an external device which presents two bytes, the first byte presented, which is the first (low addressed byte) of the two byte record is called Master, and the second byte is called Slave.

The left most (most significant) bit is called Control for each byte, and the remaining 7 bits are called V1, V2, . . . V7, where V7 is the rightmost (least significant) bit.

On a big-endian machine, we can write the following representation clause

```
type Data is record
   Master_Control : Bit;
   Master_V1      : Bit;
   Master_V2      : Bit;
   Master_V3      : Bit;
   Master_V4      : Bit;
   Master_V5      : Bit;
   Master_V6      : Bit;
   Master_V7      : Bit;
   Slave_Control  : Bit;
   Slave_V1       : Bit;
   Slave_V2       : Bit;
   Slave_V3       : Bit;
   Slave_V4       : Bit;
   Slave_V5       : Bit;
   Slave_V6       : Bit;
   Slave_V7       : Bit;
end record;

for Data use record
   Master_Control at 0 range 0 .. 0;
   Master_V1      at 0 range 1 .. 1;
```

```
      Master_V2      at 0 range 2 .. 2;
      Master_V3      at 0 range 3 .. 3;
      Master_V4      at 0 range 4 .. 4;
      Master_V5      at 0 range 5 .. 5;
      Master_V6      at 0 range 6 .. 6;
      Master_V7      at 0 range 7 .. 7;
      Slave_Control  at 1 range 0 .. 0;
      Slave_V1       at 1 range 1 .. 1;
      Slave_V2       at 1 range 2 .. 2;
      Slave_V3       at 1 range 3 .. 3;
      Slave_V4       at 1 range 4 .. 4;
      Slave_V5       at 1 range 5 .. 5;
      Slave_V6       at 1 range 6 .. 6;
      Slave_V7       at 1 range 7 .. 7;
   end record;
```

Now if we move this to a little endian machine, then the bit ordering within the byte is backwards, so we have to rewrite the record rep clause as:

```
   for Data use record
      Master_Control at 0 range 7 .. 7;
      Master_V1      at 0 range 6 .. 6;
      Master_V2      at 0 range 5 .. 5;
      Master_V3      at 0 range 4 .. 4;
      Master_V4      at 0 range 3 .. 3;
      Master_V5      at 0 range 2 .. 2;
      Master_V6      at 0 range 1 .. 1;
      Master_V7      at 0 range 0 .. 0;
      Slave_Control  at 1 range 7 .. 7;
      Slave_V1       at 1 range 6 .. 6;
      Slave_V2       at 1 range 5 .. 5;
      Slave_V3       at 1 range 4 .. 4;
      Slave_V4       at 1 range 3 .. 3;
      Slave_V5       at 1 range 2 .. 2;
      Slave_V6       at 1 range 1 .. 1;
      Slave_V7       at 1 range 0 .. 0;
   end record;
```

It is a nuisance to have to rewrite the clause, especially if the code has to be maintained on both machines. However, this is a case that we can handle with the `Bit_Order` attribute if it is implemented. Note that the implementation is not required on byte addressed machines, but it is indeed implemented in GNAT. This means that we can simply use the first record clause, together with the declaration

```
   for Data'Bit_Order use High_Order_First;
```

and the effect is what is desired, namely the layout is exactly the same, independent of whether the code is compiled on a big-endian or little-endian machine.

The important point to understand is that byte ordering is not affected. A `Bit_Order` attribute definition never affects which byte a field ends up in, only where it ends up in that byte. To make this clear, let us rewrite the record rep clause of the previous example as:

```
for Data'Bit_Order use High_Order_First;
for Data use record
   Master_Control at 0 range  0 .. 0;
   Master_V1      at 0 range  1 .. 1;
   Master_V2      at 0 range  2 .. 2;
   Master_V3      at 0 range  3 .. 3;
   Master_V4      at 0 range  4 .. 4;
   Master_V5      at 0 range  5 .. 5;
   Master_V6      at 0 range  6 .. 6;
   Master_V7      at 0 range  7 .. 7;
   Slave_Control  at 0 range  8 .. 8;
   Slave_V1       at 0 range  9 .. 9;
   Slave_V2       at 0 range 10 .. 10;
   Slave_V3       at 0 range 11 .. 11;
   Slave_V4       at 0 range 12 .. 12;
   Slave_V5       at 0 range 13 .. 13;
   Slave_V6       at 0 range 14 .. 14;
   Slave_V7       at 0 range 15 .. 15;
end record;
```

This is exactly equivalent to saying (a repeat of the first example):

```
for Data'Bit_Order use High_Order_First;
for Data use record
   Master_Control at 0 range 0 .. 0;
   Master_V1      at 0 range 1 .. 1;
   Master_V2      at 0 range 2 .. 2;
   Master_V3      at 0 range 3 .. 3;
   Master_V4      at 0 range 4 .. 4;
   Master_V5      at 0 range 5 .. 5;
   Master_V6      at 0 range 6 .. 6;
   Master_V7      at 0 range 7 .. 7;
   Slave_Control  at 1 range 0 .. 0;
   Slave_V1       at 1 range 1 .. 1;
   Slave_V2       at 1 range 2 .. 2;
   Slave_V3       at 1 range 3 .. 3;
   Slave_V4       at 1 range 4 .. 4;
   Slave_V5       at 1 range 5 .. 5;
   Slave_V6       at 1 range 6 .. 6;
   Slave_V7       at 1 range 7 .. 7;
end record;
```

Why are they equivalent? Well take a specific field, the `Slave_V2` field. The storage place attributes are obtained by normalizing the values given so that the `First_Bit` value is less than 8. After normalizing the values (0,10,10) we get (1,2,2) which is exactly what we specified in the other case.

Now one might expect that the `Bit_Order` attribute might affect bit numbering within the entire record component (two bytes in this case, thus affecting which byte fields end up in), but that is not the way this feature is defined, it only affects numbering of bits, not which byte they end up in.

Consequently it never makes sense to specify a starting bit number greater than 7 (for a byte addressable field) if an attribute definition for `Bit_Order` has been given, and indeed it may be actively confusing to specify such a value, so the compiler generates a warning for such usage.

If you do need to control byte ordering then appropriate conditional values must be used. If in our example, the slave byte came first on some machines we might write:

```
Master_Byte_First constant Boolean := ...;

Master_Byte : constant Natural :=
                1 - Boolean'Pos (Master_Byte_First);
Slave_Byte  : constant Natural :=
                Boolean'Pos (Master_Byte_First);

for Data'Bit_Order use High_Order_First;
for Data use record
   Master_Control at Master_Byte range 0 .. 0;
   Master_V1      at Master_Byte range 1 .. 1;
   Master_V2      at Master_Byte range 2 .. 2;
   Master_V3      at Master_Byte range 3 .. 3;
   Master_V4      at Master_Byte range 4 .. 4;
   Master_V5      at Master_Byte range 5 .. 5;
   Master_V6      at Master_Byte range 6 .. 6;
   Master_V7      at Master_Byte range 7 .. 7;
   Slave_Control  at Slave_Byte  range 0 .. 0;
   Slave_V1       at Slave_Byte  range 1 .. 1;
   Slave_V2       at Slave_Byte  range 2 .. 2;
   Slave_V3       at Slave_Byte  range 3 .. 3;
   Slave_V4       at Slave_Byte  range 4 .. 4;
   Slave_V5       at Slave_Byte  range 5 .. 5;
   Slave_V6       at Slave_Byte  range 6 .. 6;
   Slave_V7       at Slave_Byte  range 7 .. 7;
end record;
```

Now to switch between machines, all that is necessary is to set the boolean constant `Master_Byte_First` in an appropriate manner.

9.10 Pragma Pack for Arrays

Pragma `Pack` applied to an array has an effect that depends upon whether the component type is 'packable'. For a component type to be 'packable', it must be one of the following cases:

* Any elementary type.
* Any small packed array type with a static size.
* Any small simple record type with a static size.

For all these cases, if the component subtype size is in the range 1 through 63 on 32-bit targets, and 1 through 127 on 64-bit targets, then the effect of the pragma `Pack` is exactly as though a component size were specified giving the component subtype size.

All other types are non-packable, they occupy an integral number of storage units and the only effect of pragma Pack is to remove alignment gaps.

For example if we have:

```
type r is range 0 .. 17;

type ar is array (1 .. 8) of r;
pragma Pack (ar);
```

Then the component size of `ar` will be set to 5 (i.e., to `r'size`, and the size of the array `ar` will be exactly 40 bits).

Note that in some cases this rather fierce approach to packing can produce unexpected effects. For example, in Ada 95 and Ada 2005, subtype `Natural` typically has a size of 31, meaning that if you pack an array of `Natural`, you get 31-bit close packing, which saves a few bits, but results in far less efficient access. Since many other Ada compilers will ignore such a packing request, GNAT will generate a warning on some uses of pragma `Pack` that it guesses might not be what is intended. You can easily remove this warning by using an explicit `Component_Size` setting instead, which never generates a warning, since the intention of the programmer is clear in this case.

GNAT treats packed arrays in one of two ways. If the size of the array is known at compile time and is at most 64 bits on 32-bit targets, and at most 128 bits on 64-bit targets, then internally the array is represented as a single modular type, of exactly the appropriate number of bits. If the length is greater than 64 bits on 32-bit targets, and greater than 128 bits on 64-bit targets, or is not known at compile time, then the packed array is represented as an array of bytes, and its length is always a multiple of 8 bits.

Note that to represent a packed array as a modular type, the alignment must be suitable for the modular type involved. For example, on typical machines a 32-bit packed array will be represented by a 32-bit modular integer with an alignment of four bytes. If you explicitly override the default alignment with an alignment clause that is too small, the modular representation cannot be used. For example, consider the following set of declarations:

```
type R is range 1 .. 3;
type S is array (1 .. 31) of R;
for S'Component_Size use 2;
for S'Size use 62;
for S'Alignment use 1;
```

If the alignment clause were not present, then a 62-bit modular representation would be chosen (typically with an alignment of 4 or 8 bytes depending on the target). But the default alignment is overridden with the explicit alignment clause. This means that the modular representation cannot be used, and instead the array of bytes representation must be used, meaning that the length must be a multiple of 8. Thus the above set of declarations will result in a diagnostic rejecting the size clause and noting that the minimum size allowed is 64.

One special case that is worth noting occurs when the base type of the component size is 8/16/32 and the subtype is one bit less. Notably this occurs with subtype `Natural`. Consider:

```
type Arr is array (1 .. 32) of Natural;
pragma Pack (Arr);
```

In all commonly used Ada 83 compilers, this pragma Pack would be ignored, since typically `Natural'Size` is 32 in Ada 83, and in any case most Ada 83 compilers did not attempt 31 bit packing.

In Ada 95 and Ada 2005, `Natural'Size` is required to be 31. Furthermore, GNAT really does pack 31-bit subtype to 31 bits. This may result in a substantial unintended performance penalty when porting legacy Ada 83 code. To help prevent this, GNAT generates a warning in such cases. If you really want 31 bit packing in a case like this, you can set the component size explicitly:

```
type Arr is array (1 .. 32) of Natural;
for Arr'Component_Size use 31;
```

Here 31-bit packing is achieved as required, and no warning is generated, since in this case the programmer intention is clear.

9.11 Pragma Pack for Records

Pragma `Pack` applied to a record will pack the components to reduce wasted space from alignment gaps and by reducing the amount of space taken by components. We distinguish between 'packable' components and 'non-packable' components. Components of the following types are considered packable:

* Components of an elementary type are packable unless they are aliased, independent or atomic.
* Small packed arrays, where the size is statically known, are represented internally as modular integers, and so they are also packable.
* Small simple records, where the size is statically known, are also packable.

For all these cases, if the `'Size` value is in the range 1 through 64 on 32-bit targets, and 1 through 128 on 64-bit targets, the components occupy the exact number of bits corresponding to this value and are packed with no padding bits, i.e. they can start on an arbitrary bit boundary.

All other types are non-packable, they occupy an integral number of storage units and the only effect of pragma `Pack` is to remove alignment gaps.

For example, consider the record

```
type Rb1 is array (1 .. 13) of Boolean;
pragma Pack (Rb1);

type Rb2 is array (1 .. 65) of Boolean;
pragma Pack (Rb2);

type AF is new Float with Atomic;
```

```
type X2 is record
   L1 : Boolean;
   L2 : Duration;
   L3 : AF;
   L4 : Boolean;
   L5 : Rb1;
   L6 : Rb2;
end record;
pragma Pack (X2);
```

The representation for the record X2 is as follows on 32-bit targets:

```
for X2'Size use 224;
for X2 use record
   L1 at  0 range  0 .. 0;
   L2 at  0 range  1 .. 64;
   L3 at 12 range  0 .. 31;
   L4 at 16 range  0 .. 0;
   L5 at 16 range  1 .. 13;
   L6 at 18 range  0 .. 71;
end record;
```

Studying this example, we see that the packable fields L1 and L2 are of length equal to their sizes, and placed at specific bit boundaries (and not byte boundaries) to eliminate padding. But L3 is of a non-packable float type (because it is aliased), so it is on the next appropriate alignment boundary.

The next two fields are fully packable, so L4 and L5 are minimally packed with no gaps. However, type Rb2 is a packed array that is longer than 64 bits, so it is itself non-packable on 32-bit targets. Thus the L6 field is aligned to the next byte boundary, and takes an integral number of bytes, i.e., 72 bits.

9.12 Record Representation Clauses

Record representation clauses may be given for all record types, including types obtained by record extension. Component clauses are allowed for any static component. The restrictions on component clauses depend on the type of the component.

For all components of an elementary type, the only restriction on component clauses is that the size must be at least the 'Size value of the type (actually the Value_Size). There are no restrictions due to alignment, and such components may freely cross storage boundaries.

Packed arrays with a size up to and including 64 bits on 32-bit targets, and up to and including 128 bits on 64-bit targets, are represented internally using a modular type with the appropriate number of bits, and thus the same lack of restriction applies. For example, if you declare:

```
type R is array (1 .. 49) of Boolean;
pragma Pack (R);
for R'Size use 49;
```

then a component clause for a component of type R may start on any specified bit boundary, and may specify a value of 49 bits or greater.

For packed bit arrays that are longer than 64 bits on 32-bit targets, and longer than 128 bits on 64-bit targets, there are two cases. If the component size is a power of 2 (1,2,4,8,16,32,64 bits), including the important case of single bits or boolean values, then there are no limitations on placement of such components, and they may start and end at arbitrary bit boundaries.

If the component size is not a power of 2 (e.g., 3 or 5), then an array of this type must always be placed on on a storage unit (byte) boundary and occupy an integral number of storage units (bytes). Any component clause that does not meet this requirement will be rejected.

Any aliased component, or component of an aliased type, must have its normal alignment and size. A component clause that does not meet this requirement will be rejected.

The tag field of a tagged type always occupies an address sized field at the start of the record. No component clause may attempt to overlay this tag. When a tagged type appears as a component, the tag field must have proper alignment

In the case of a record extension `T1`, of a type `T`, no component clause applied to the type `T1` can specify a storage location that would overlap the first `T'Object_Size` bits of the record.

For all other component types, including non-bit-packed arrays, the component can be placed at an arbitrary bit boundary, so for example, the following is permitted:

```
type R is array (1 .. 10) of Boolean;
for R'Size use 80;

type Q is record
   G, H : Boolean;
   L, M : R;
end record;

for Q use record
   G at 0 range  0 ..   0;
   H at 0 range  1 ..   1;
   L at 0 range  2 ..  81;
   R at 0 range 82 .. 161;
end record;
```

9.13 Handling of Records with Holes

As a result of alignment considerations, records may contain "holes" or gaps which do not correspond to the data bits of any of the components. Record representation clauses can also result in holes in records.

GNAT does not attempt to clear these holes, so in record objects, they should be considered to hold undefined rubbish. The generated equality routine just tests components so does not access these undefined bits, and assignment and copy operations may or may not preserve the contents of these holes (for assignments, the holes in the target will in practice contain either the bits that are present in the holes in the source, or the bits that were present in the target before the assignment).

If it is necessary to ensure that holes in records have all zero bits, then record objects for which this initialization is desired should be explicitly set to all zero values using Unchecked_Conversion or address overlays. For example

```
type HRec is record
   C : Character;
   I : Integer;
end record;
```

On typical machines, integers need to be aligned on a four-byte boundary, resulting in three bytes of undefined rubbish following the 8-bit field for C. To ensure that the hole in a variable of type HRec is set to all zero bits, you could for example do:

```
type Base is record
   Dummy1, Dummy2 : Integer := 0;
end record;

BaseVar : Base;
RealVar : Hrec;
for RealVar'Address use BaseVar'Address;
```

Now the 8-bytes of the value of RealVar start out containing all zero bits. A safer approach is to just define dummy fields, avoiding the holes, as in:

```
type HRec is record
   C      : Character;
   Dummy1 : Short_Short_Integer := 0;
   Dummy2 : Short_Short_Integer := 0;
   Dummy3 : Short_Short_Integer := 0;
   I      : Integer;
end record;
```

And to make absolutely sure that the intent of this is followed, you can use representation clauses:

```
for Hrec use record
   C      at 0 range 0 .. 7;
   Dummy1 at 1 range 0 .. 7;
   Dummy2 at 2 range 0 .. 7;
   Dummy3 at 3 range 0 .. 7;
   I      at 4 range 0 .. 31;
end record;
for Hrec'Size use 64;
```

9.14 Enumeration Clauses

The only restriction on enumeration clauses is that the range of values must be representable. For the signed case, if one or more of the representation values are negative, all values must be in the range:

```
System.Min_Int .. System.Max_Int
```

For the unsigned case, where all values are nonnegative, the values must be in the range:

```
0 .. System.Max_Binary_Modulus;
```

A 'confirming' representation clause is one in which the values range from 0 in sequence, i.e., a clause that confirms the default representation for an enumeration type. Such a confirming representation is permitted by these rules, and is specially recognized by the compiler so that no extra overhead results from the use of such a clause.

If an array has an index type which is an enumeration type to which an enumeration clause has been applied, then the array is stored in a compact manner. Consider the declarations:

```
type r is (A, B, C);
for r use (A => 1, B => 5, C => 10);
type t is array (r) of Character;
```

The array type t corresponds to a vector with exactly three elements and has a default size equal to `3*Character'Size`. This ensures efficient use of space, but means that accesses to elements of the array will incur the overhead of converting representation values to the corresponding positional values, (i.e., the value delivered by the `Pos` attribute).

9.15 Address Clauses

The reference manual allows a general restriction on representation clauses, as found in RM 13.1(22):

> "An implementation need not support representation items containing nonstatic expressions, except that an implementation should support a representation item for a given entity if each nonstatic expression in the representation item is a name that statically denotes a constant declared before the entity."

In practice this is applicable only to address clauses, since this is the only case in which a nonstatic expression is permitted by the syntax. As the AARM notes in sections 13.1 (22.a-22.h):

> 22.a Reason: This is to avoid the following sort of thing:
>
> 22.b X : Integer := F(. . .); Y : Address := G(. . .); for X'Address use Y;
>
> 22.c In the above, we have to evaluate the initialization expression for X before we know where to put the result. This seems like an unreasonable implementation burden.
>
> 22.d The above code should instead be written like this:
>
> 22.e Y : constant Address := G(. . .); X : Integer := F(. . .); for X'Address use Y;
>
> 22.f This allows the expression 'Y' to be safely evaluated before X is created.
>
> 22.g The constant could be a formal parameter of mode in.
>
> 22.h An implementation can support other nonstatic expressions if it wants to. Expressions of type Address are hardly ever static, but their value might be known at compile time anyway in many cases.

GNAT does indeed permit many additional cases of nonstatic expressions. In particular, if the type involved is elementary there are no restrictions (since in this case, holding a temporary copy of the initialization value, if one is present, is inexpensive). In addition, if there is no implicit or explicit initialization, then there are no restrictions. GNAT will reject only the case where all three of these conditions hold:

* The type of the item is non-elementary (e.g., a record or array).
* There is explicit or implicit initialization required for the object. Note that access values are always implicitly initialized.
* The address value is nonstatic. Here GNAT is more permissive than the RM, and allows the address value to be the address of a previously declared stand-alone variable, as long as it does not itself have an address clause.

  ```
  Anchor  : Some_Initialized_Type;
  Overlay : Some_Initialized_Type;
  for Overlay'Address use Anchor'Address;
  ```

  However, the prefix of the address clause cannot be an array component, or a component of a discriminated record.

As noted above in section 22.h, address values are typically nonstatic. In particular the To_Address function, even if applied to a literal value, is a nonstatic function call. To avoid this minor annoyance, GNAT provides the implementation defined attribute 'To_Address. The following two expressions have identical values:

```
To_Address (16#1234_0000#)
System'To_Address (16#1234_0000#);
```

except that the second form is considered to be a static expression, and thus when used as an address clause value is always permitted.

Additionally, GNAT treats as static an address clause that is an unchecked_conversion of a static integer value. This simplifies the porting of legacy code, and provides a portable equivalent to the GNAT attribute `To_Address`.

Another issue with address clauses is the interaction with alignment requirements. When an address clause is given for an object, the address value must be consistent with the alignment of the object (which is usually the same as the alignment of the type of the object). If an address clause is given that specifies an inappropriately aligned address value, then the program execution is erroneous.

Since this source of erroneous behavior can have unfortunate effects on machines with strict alignment requirements, GNAT checks (at compile time if possible, generating a warning, or at execution time with a run-time check) that the alignment is appropriate. If the run-time check fails, then `Program_Error` is raised. This run-time check is suppressed if the GNAT check Alignment_Check is suppressed, or if `pragma Restrictions (No_Elaboration_Code)` is in effect. It is also suppressed by default on non-strict alignment machines (such as the x86).

In some cases, GNAT does not support an address specification (using either form of aspect specification syntax) for the declaration of an object that has an indefinite nominal subtype. An object declaration has an indefinite nominal subtype if it takes its bounds (for an array type), discriminant values (for a discriminated type whose discriminants lack defaults), or tag (for a class-wide type) from its initial value, as in

```
X : String := Some_Function_Call;
-- String has no constraint, so bounds for X come from function call
```

This restriction does not apply if the size of the object's initial value is known at compile time and the type of the object is not class-wide.

An address clause cannot be given for an exported object. More understandably the real restriction is that objects with an address clause cannot be exported. This is because such variables are not defined by the Ada program, so there is no external object to export.

It is permissible to give an address clause and a pragma Import for the same object. In this case, the variable is not really defined by the Ada program, so there is no external symbol to be linked. The link name and the external name are ignored in this case. The reason that we allow this combination is that it provides a useful idiom to avoid unwanted initializations on objects with address clauses.

When an address clause is given for an object that has implicit or explicit initialization, then by default initialization takes place. This means that the effect of the object declaration is to overwrite the memory at the specified address. This is almost always not what the programmer wants, so GNAT will output a warning:

```
with System;
package G is
   type R is record
      M : Integer := 0;
   end record;

   Ext : R;
   for Ext'Address use System'To_Address (16#1234_1234#);
       |
>>> warning: implicit initialization of "Ext" may
    modify overlaid storage
>>> warning: use pragma Import for "Ext" to suppress
    initialization (RM B(24))

end G;
```

As indicated by the warning message, the solution is to use a (dummy) pragma Import to suppress this initialization. The pragma tell the compiler that the object is declared and initialized elsewhere. The following package compiles without warnings (and the initialization is suppressed):

```
with System;
package G is
   type R is record
      M : Integer := 0;
   end record;

   Ext : R;
   for Ext'Address use System'To_Address (16#1234_1234#);
   pragma Import (Ada, Ext);
end G;
```

A final issue with address clauses involves their use for overlaying variables, as in the following example:

```
A : Integer;
B : Integer;
for B'Address use A'Address;
```

or alternatively, using the form recommended by the RM:

```
A    : Integer;
Addr : constant Address := A'Address;
B    : Integer;
for B'Address use Addr;
```

In both of these cases, `A` and `B` become aliased to one another via the address clause. This use of address clauses to overlay variables, achieving an effect similar to unchecked conversion was erroneous in Ada 83, but in Ada 95 and Ada 2005 the effect is implementation defined. Furthermore, the Ada RM specifically recommends that in a situation like this, `B` should be subject to the following implementation advice (RM 13.3(19)):

> "19 If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases."

GNAT follows this recommendation, and goes further by also applying this recommendation to the overlaid variable (`A` in the above example) in this case. This means that the overlay works "as expected", in that a modification to one of the variables will affect the value of the other.

More generally, GNAT interprets this recommendation conservatively for address clauses: in the cases other than overlays, it considers that the object is effectively subject to pragma `Volatile` and implements the associated semantics.

Note that when address clause overlays are used in this way, there is an issue of unintentional initialization, as shown by this example:

```
package Overwrite_Record is
   type R is record
      A : Character := 'C';
      B : Character := 'A';
   end record;
   X : Short_Integer := 3;
   Y : R;
   for Y'Address use X'Address;
       |
>>> warning: default initialization of "Y" may
    modify "X", use pragma Import for "Y" to
    suppress initialization (RM B.1(24))

end Overwrite_Record;
```

Here the default initialization of `Y` will clobber the value of `X`, which justifies the warning. The warning notes that this effect can be eliminated by adding a `pragma Import` which suppresses the initialization:

```
package Overwrite_Record is
   type R is record
      A : Character := 'C';
```

```
      B : Character := 'A';
   end record;
   X : Short_Integer := 3;
   Y : R;
   for Y'Address use X'Address;
   pragma Import (Ada, Y);
end Overwrite_Record;
```

Note that the use of `pragma Initialize_Scalars` may cause variables to be initialized when they would not otherwise have been in the absence of the use of this pragma. This may cause an overlay to have this unintended clobbering effect. The compiler avoids this for scalar types, but not for composite objects (where in general the effect of `Initialize_Scalars` is part of the initialization routine for the composite object):

```
pragma Initialize_Scalars;
with Ada.Text_IO;  use Ada.Text_IO;
procedure Overwrite_Array is
   type Arr is array (1 .. 5) of Integer;
   X : Arr := (others => 1);
   A : Arr;
   for A'Address use X'Address;
       |
>>> warning: default initialization of "A" may
    modify "X", use pragma Import for "A" to
    suppress initialization (RM B.1(24))

begin
   if X /= Arr'(others => 1) then
      Put_Line ("X was clobbered");
   else
      Put_Line ("X was not clobbered");
   end if;
end Overwrite_Array;
```

The above program generates the warning as shown, and at execution time, prints `X was clobbered`. If the `pragma Import` is added as suggested:

```
pragma Initialize_Scalars;
with Ada.Text_IO;  use Ada.Text_IO;
procedure Overwrite_Array is
   type Arr is array (1 .. 5) of Integer;
   X : Arr := (others => 1);
   A : Arr;
   for A'Address use X'Address;
   pragma Import (Ada, A);
begin
   if X /= Arr'(others => 1) then
      Put_Line ("X was clobbered");
   else
      Put_Line ("X was not clobbered");
```

```
   end if;
end Overwrite_Array;
```

then the program compiles without the warning and when run will generate the output `X was not clobbered`.

9.16 Use of Address Clauses for Memory-Mapped I/O

A common pattern is to use an address clause to map an atomic variable to a location in memory that corresponds to a memory-mapped I/O operation or operations, for example:

```
type Mem_Word is record
   A,B,C,D : Byte;
end record;
pragma Atomic (Mem_Word);
for Mem_Word_Size use 32;

Mem : Mem_Word;
for Mem'Address use some-address;
...
Temp := Mem;
Temp.A := 32;
Mem := Temp;
```

For a full access (reference or modification) of the variable (Mem) in this case, as in the above examples, GNAT guarantees that the entire atomic word will be accessed, in accordance with the RM C.6(15) clause.

A problem arises with a component access such as:

```
Mem.A := 32;
```

Note that the component A is not declared as atomic. This means that it is not clear what this assignment means. It could correspond to full word read and write as given in the first example, or on architectures that supported such an operation it might be a single byte store instruction. The RM does not have anything to say in this situation, and GNAT does not make any guarantee. The code generated may vary from target to target. GNAT will issue a warning in such a case:

```
Mem.A := 32;
|
>>> warning: access to non-atomic component of atomic array,
    may cause unexpected accesses to atomic object
```

It is best to be explicit in this situation, by either declaring the components to be atomic if you want the byte store, or explicitly writing the full word access sequence if that is what the hardware requires. Alternatively, if the full word access sequence is required, GNAT also provides the pragma `Volatile_Full_Access` which can be used in lieu of pragma `Atomic` and will give the additional guarantee.

9.17 Effect of Convention on Representation

Normally the specification of a foreign language convention for a type or an object has no effect on the chosen representation. In particular, the representation chosen for data in

GNAT generally meets the standard system conventions, and for example records are laid out in a manner that is consistent with C. This means that specifying convention C (for example) has no effect.

There are three exceptions to this general rule:

* ‘Convention Fortran and array subtypes’.

  If pragma Convention Fortran is specified for an array subtype, then in accordance with the implementation advice in section 3.6.2(11) of the Ada Reference Manual, the array will be stored in a Fortran-compatible column-major manner, instead of the normal default row-major order.

* ‘Convention C and enumeration types’

  GNAT normally stores enumeration types in 8, 16, or 32 bits as required to accommodate all values of the type. For example, for the enumeration type declared by:

  ```
  type Color is (Red, Green, Blue);
  ```

  8 bits is sufficient to store all values of the type, so by default, objects of type `Color` will be represented using 8 bits. However, normal C convention is to use 32 bits for all enum values in C, since enum values are essentially of type int. If pragma `Convention C` is specified for an Ada enumeration type, then the size is modified as necessary (usually to 32 bits) to be consistent with the C convention for enum values.

  Note that this treatment applies only to types. If Convention C is given for an enumeration object, where the enumeration type is not Convention C, then Object_Size bits are allocated. For example, for a normal enumeration type, with less than 256 elements, only 8 bits will be allocated for the object. Since this may be a surprise in terms of what C expects, GNAT will issue a warning in this situation. The warning can be suppressed by giving an explicit size clause specifying the desired size.

* ‘Convention C/Fortran and Boolean types’

  In C, the usual convention for boolean values, that is values used for conditions, is that zero represents false, and nonzero values represent true. In Ada, the normal convention is that two specific values, typically 0/1, are used to represent false/true respectively.

  Fortran has a similar convention for `LOGICAL` values (any nonzero value represents true).

  To accommodate the Fortran and C conventions, if a pragma Convention specifies C or Fortran convention for a derived Boolean, as in the following example:

  ```
  type C_Switch is new Boolean;
  pragma Convention (C, C_Switch);
  ```

  then the GNAT generated code will treat any nonzero value as true. For truth values generated by GNAT, the conventional value 1 will be used for True, but when one of these values is read, any nonzero value is treated as True.

9.18 Conventions and Anonymous Access Types

The RM is not entirely clear on convention handling in a number of cases, and in particular, it is not clear on the convention to be given to anonymous access types in general, and in particular what is to be done for the case of anonymous access-to-subprogram.

In GNAT, we decide that if an explicit Convention is applied to an object or component, and its type is such an anonymous type, then the convention will apply to this anonymous type as well. This seems to make sense since it is anomolous in any case to have a different convention for an object and its type, and there is clearly no way to explicitly specify a convention for an anonymous type, since it doesn't have a name to specify!

Furthermore, we decide that if a convention is applied to a record type, then this convention is inherited by any of its components that are of an anonymous access type which do not have an explicitly specified convention.

The following program shows these conventions in action:

```
package ConvComp is
   type Foo is range 1 .. 10;
   type T1 is record
      A : access function (X : Foo) return Integer;
      B : Integer;
   end record;
   pragma Convention (C, T1);

   type T2 is record
      A : access function (X : Foo) return Integer;
      pragma Convention  (C, A);
      B : Integer;
   end record;
   pragma Convention (COBOL, T2);

   type T3 is record
      A : access function (X : Foo) return Integer;
      pragma Convention  (COBOL, A);
      B : Integer;
   end record;
   pragma Convention (C, T3);

   type T4 is record
      A : access function (X : Foo) return Integer;
      B : Integer;
   end record;
   pragma Convention (COBOL, T4);

   function F (X : Foo) return Integer;
   pragma Convention (C, F);

   function F (X : Foo) return Integer is (13);

   TV1 : T1 := (F'Access, 12);  -- OK
   TV2 : T2 := (F'Access, 13);  -- OK

   TV3 : T3 := (F'Access, 13);  -- ERROR
```

```
                |
>>> subprogram "F" has wrong convention
>>> does not match access to subprogram declared at line 17
     38.    TV4 : T4 := (F'Access, 13);  -- ERROR
                |
>>> subprogram "F" has wrong convention
>>> does not match access to subprogram declared at line 24
     39. end ConvComp;
```

9.19 Determining the Representations chosen by GNAT

Although the descriptions in this section are intended to be complete, it is often easier to simply experiment to see what GNAT accepts and what the effect is on the layout of types and objects.

As required by the Ada RM, if a representation clause is not accepted, then it must be rejected as illegal by the compiler. However, when a representation clause or pragma is accepted, there can still be questions of what the compiler actually does. For example, if a partial record representation clause specifies the location of some components and not others, then where are the non-specified components placed? Or if pragma `Pack` is used on a record, then exactly where are the resulting fields placed? The section on pragma `Pack` in this chapter can be used to answer the second question, but it is often easier to just see what the compiler does.

For this purpose, GNAT provides the option '-gnatR'. If you compile with this option, then the compiler will output information on the actual representations chosen, in a format similar to source representation clauses. For example, if we compile the package:

```
package q is
   type r (x : boolean) is tagged record
      case x is
         when True => S : String (1 .. 100);
         when False => null;
      end case;
   end record;

   type r2 is new r (false) with record
      y2 : integer;
   end record;

   for r2 use record
      y2 at 16 range 0 .. 31;
   end record;

   type x is record
      y : character;
   end record;

   type x1 is array (1 .. 10) of x;
   for x1'component_size use 11;
```

```
   type ia is access integer;

   type Rb1 is array (1 .. 13) of Boolean;
   pragma Pack (rb1);

   type Rb2 is array (1 .. 65) of Boolean;
   pragma Pack (rb2);

   type x2 is record
      l1 : Boolean;
      l2 : Duration;
      l3 : Float;
      l4 : Boolean;
      l5 : Rb1;
      l6 : Rb2;
   end record;
   pragma Pack (x2);
end q;
```

using the switch '-gnatR' we obtain the following output:

```
Representation information for unit q
-------------------------------------

for r'Size use ??;
for r'Alignment use 4;
for r use record
   x    at 4 range  0 .. 7;
   _tag at 0 range  0 .. 31;
   s    at 5 range  0 .. 799;
end record;

for r2'Size use 160;
for r2'Alignment use 4;
for r2 use record
   x       at  4 range  0 .. 7;
   _tag    at  0 range  0 .. 31;
   _parent at  0 range  0 .. 63;
   y2      at 16 range  0 .. 31;
end record;

for x'Size use 8;
for x'Alignment use 1;
for x use record
   y at 0 range  0 .. 7;
end record;
```

```
for x1'Size use 112;
for x1'Alignment use 1;
for x1'Component_Size use 11;

for rb1'Size use 13;
for rb1'Alignment use 2;
for rb1'Component_Size use 1;

for rb2'Size use 72;
for rb2'Alignment use 1;
for rb2'Component_Size use 1;

for x2'Size use 224;
for x2'Alignment use 4;
for x2 use record
   l1 at  0 range  0 .. 0;
   l2 at  0 range  1 .. 64;
   l3 at 12 range  0 .. 31;
   l4 at 16 range  0 .. 0;
   l5 at 16 range  1 .. 13;
   l6 at 18 range  0 .. 71;
end record;
```

The Size values are actually the Object_Size, i.e., the default size that will be allocated for objects of the type. The `??` size for type r indicates that we have a variant record, and the actual size of objects will depend on the discriminant value.

The Alignment values show the actual alignment chosen by the compiler for each record or array type.

The record representation clause for type r shows where all fields are placed, including the compiler generated tag field (whose location cannot be controlled by the programmer).

The record representation clause for the type extension r2 shows all the fields present, including the parent field, which is a copy of the fields of the parent type of r2, i.e., r1.

The component size and size clauses for types rb1 and rb2 show the exact effect of pragma `Pack` on these arrays, and the record representation clause for type x2 shows how pragma *Pack* affects this record type.

In some cases, it may be useful to cut and paste the representation clauses generated by the compiler into the original source to fix and guarantee the actual representation to be used.

10 Standard Library Routines

The Ada Reference Manual contains in Annex A a full description of an extensive set of standard library routines that can be used in any Ada program, and which must be provided by all Ada compilers. They are analogous to the standard C library used by C programs.

GNAT implements all of the facilities described in annex A, and for most purposes the description in the Ada Reference Manual, or appropriate Ada text book, will be sufficient for making use of these facilities.

In the case of the input-output facilities, [The Implementation of Standard I/O], page 243, gives details on exactly how GNAT interfaces to the file system. For the remaining packages, the Ada Reference Manual should be sufficient. The following is a list of the packages included, together with a brief description of the functionality that is provided.

For completeness, references are included to other predefined library routines defined in other sections of the Ada Reference Manual (these are cross-indexed from Annex A). For further details see the relevant package declarations in the run-time library. In particular, a few units are not implemented, as marked by the presence of pragma Unimplemented_Unit, and in this case the package declaration contains comments explaining why the unit is not implemented.

`Ada` '(A.2)'
: This is a parent package for all the standard library packages. It is usually included implicitly in your program, and itself contains no useful data or routines.

`Ada.Assertions` '(11.4.2)'
: `Assertions` provides the `Assert` subprograms, and also the declaration of the `Assertion_Error` exception.

`Ada.Asynchronous_Task_Control` '(D.11)'
: `Asynchronous_Task_Control` provides low level facilities for task synchronization. It is typically not implemented. See package spec for details.

`Ada.Calendar` '(9.6)'
: `Calendar` provides time of day access, and routines for manipulating times and durations.

`Ada.Calendar.Arithmetic` '(9.6.1)'
: This package provides additional arithmetic operations for `Calendar`.

`Ada.Calendar.Formatting` '(9.6.1)'
: This package provides formatting operations for `Calendar`.

`Ada.Calendar.Time_Zones` '(9.6.1)'
: This package provides additional `Calendar` facilities for handling time zones.

`Ada.Characters` '(A.3.1)'
: This is a dummy parent package that contains no useful entities

`Ada.Characters.Conversions` '(A.3.2)'
: This package provides character conversion functions.

`Ada.Characters.Handling` '(A.3.2)'
: This package provides some basic character handling capabilities, including classification functions for classes of characters (e.g., test for letters, or digits).

`Ada.Characters.Latin_1` '(A.3.3)'
: This package includes a complete set of definitions of the characters that appear in type CHARACTER. It is useful for writing programs that will run in international environments. For example, if you want an upper case E with an acute accent in a string, it is often better to use the definition of `UC_E_Acute` in this package. Then your program will print in an understandable manner even if your environment does not support these extended characters.

`Ada.Command_Line` '(A.15)'
: This package provides access to the command line parameters and the name of the current program (analogous to the use of `argc` and `argv` in C), and also allows the exit status for the program to be set in a system-independent manner.

`Ada.Complex_Text_IO` '(G.1.3)'
: This package provides text input and output of complex numbers.

`Ada.Containers` '(A.18.1)'
: A top level package providing a few basic definitions used by all the following specific child packages that provide specific kinds of containers.

`Ada.Containers.Bounded_Priority_Queues` '(A.18.31)'

`Ada.Containers.Bounded_Synchronized_Queues` '(A.18.29)'

`Ada.Containers.Doubly_Linked_Lists` '(A.18.3)'

`Ada.Containers.Generic_Array_Sort` '(A.18.26)'

`Ada.Containers.Generic_Constrained_Array_Sort` '(A.18.26)'

`Ada.Containers.Generic_Sort` '(A.18.26)'

`Ada.Containers.Hashed_Maps` '(A.18.5)'

`Ada.Containers.Hashed_Sets` '(A.18.8)'

`Ada.Containers.Indefinite_Doubly_Linked_Lists` '(A.18.12)'

`Ada.Containers.Indefinite_Hashed_Maps` '(A.18.13)'

`Ada.Containers.Indefinite_Hashed_Sets` '(A.18.15)'

`Ada.Containers.Indefinite_Holders` '(A.18.18)'

`Ada.Containers.Indefinite_Multiway_Trees` '(A.18.17)'

`Ada.Containers.Indefinite_Ordered_Maps` '(A.18.14)'

`Ada.Containers.Indefinite_Ordered_Sets` '(A.18.16)'

`Ada.Containers.Indefinite_Vectors` '(A.18.11)'

`Ada.Containers.Multiway_Trees` '(A.18.10)'

`Ada.Containers.Ordered_Maps` '(A.18.6)'

`Ada.Containers.Ordered_Sets` '(A.18.9)'

`Ada.Containers.Synchronized_Queue_Interfaces` '(A.18.27)'

`Ada.Containers.Unbounded_Priority_Queues` '(A.18.30)'

`Ada.Containers.Unbounded_Synchronized_Queues` '(A.18.28)'

`Ada.Containers.Vectors` '(A.18.2)'

`Ada.Directories` '(A.16)'
: This package provides operations on directories.

`Ada.Directories.Hierarchical_File_Names` '(A.16.1)'
: This package provides additional directory operations handling hierarchical file names.

`Ada.Directories.Information` '(A.16)'
: This is an implementation defined package for additional directory operations, which is not implemented in GNAT.

`Ada.Decimal` '(F.2)'
: This package provides constants describing the range of decimal numbers implemented, and also a decimal divide routine (analogous to the COBOL verb DIVIDE . . . GIVING . . . REMAINDER . . .)

`Ada.Direct_IO` '(A.8.4)'
: This package provides input-output using a model of a set of records of fixed-length, containing an arbitrary definite Ada type, indexed by an integer record number.

`Ada.Dispatching` '(D.2.1)'
: A parent package containing definitions for task dispatching operations.

`Ada.Dispatching.EDF` '(D.2.6)'
: Not implemented in GNAT.

`Ada.Dispatching.Non_Preemptive` '(D.2.4)'
: Not implemented in GNAT.

`Ada.Dispatching.Round_Robin` '(D.2.5)'
: Not implemented in GNAT.

`Ada.Dynamic_Priorities` '(D.5)'
: This package allows the priorities of a task to be adjusted dynamically as the task is running.

`Ada.Environment_Variables` '(A.17)'
: This package provides facilities for accessing environment variables.

`Ada.Exceptions` '(11.4.1)'
: This package provides additional information on exceptions, and also contains facilities for treating exceptions as data objects, and raising exceptions with associated messages.

`Ada.Execution_Time` '(D.14)'
: This package provides CPU clock functionalities. It is not implemented on all targets (see package spec for details).

`Ada.Execution_Time.Group_Budgets` '(D.14.2)'
: Not implemented in GNAT.

`Ada.Execution_Time.Timers` '(D.14.1)''
Not implemented in GNAT.

`Ada.Finalization` '(7.6)'
This package contains the declarations and subprograms to support the use of controlled types, providing for automatic initialization and finalization (analogous to the constructors and destructors of C++).

`Ada.Float_Text_IO` '(A.10.9)'
A library level instantiation of Text_IO.Float_IO for type Float.

`Ada.Float_Wide_Text_IO` '(A.10.9)'
A library level instantiation of Wide_Text_IO.Float_IO for type Float.

`Ada.Float_Wide_Wide_Text_IO` '(A.10.9)'
A library level instantiation of Wide_Wide_Text_IO.Float_IO for type Float.

`Ada.Integer_Text_IO` '(A.10.9)'
A library level instantiation of Text_IO.Integer_IO for type Integer.

`Ada.Integer_Wide_Text_IO` '(A.10.9)'
A library level instantiation of Wide_Text_IO.Integer_IO for type Integer.

`Ada.Integer_Wide_Wide_Text_IO` '(A.10.9)'
A library level instantiation of Wide_Wide_Text_IO.Integer_IO for type Integer.

`Ada.Interrupts` '(C.3.2)'
This package provides facilities for interfacing to interrupts, which includes the set of signals or conditions that can be raised and recognized as interrupts.

`Ada.Interrupts.Names` '(C.3.2)'
This package provides the set of interrupt names (actually signal or condition names) that can be handled by GNAT.

`Ada.IO_Exceptions` '(A.13)'
This package defines the set of exceptions that can be raised by use of the standard IO packages.

`Ada.Iterator_Interfaces` '(5.5.1)'
This package provides a generic interface to generalized iterators.

`Ada.Locales` '(A.19)'
This package provides declarations providing information (Language and Country) about the current locale.

`Ada.Numerics`
This package contains some standard constants and exceptions used throughout the numerics packages. Note that the constants pi and e are defined here, and it is better to use these definitions than rolling your own.

`Ada.Numerics.Complex_Arrays` '(G.3.2)'
Provides operations on arrays of complex numbers.

`Ada.Numerics.Complex_Elementary_Functions`
Provides the implementation of standard elementary functions (such as log and trigonometric functions) operating on complex numbers using the stan-

dard Float and the Complex and Imaginary types created by the package Numerics.Complex_Types.

Ada.Numerics.Complex_Types
: This is a predefined instantiation of Numerics.Generic_Complex_Types using Standard.Float to build the type Complex and Imaginary.

Ada.Numerics.Discrete_Random
: This generic package provides a random number generator suitable for generating uniformly distributed values of a specified discrete subtype. It should not be used as a cryptographic pseudo-random source.

Ada.Numerics.Float_Random
: This package provides a random number generator suitable for generating uniformly distributed floating point values in the unit interval. It should not be used as a cryptographic pseudo-random source.

Ada.Numerics.Generic_Complex_Elementary_Functions
: This is a generic version of the package that provides the implementation of standard elementary functions (such as log and trigonometric functions) for an arbitrary complex type.

  The following predefined instantiations of this package are provided:

  * Short_Float

    Ada.Numerics.Short_Complex_Elementary_Functions
  * Float

    Ada.Numerics.Complex_Elementary_Functions
  * Long_Float

    Ada.Numerics.Long_Complex_Elementary_Functions

Ada.Numerics.Generic_Complex_Types
: This is a generic package that allows the creation of complex types, with associated complex arithmetic operations.

  The following predefined instantiations of this package exist

  * Short_Float

    Ada.Numerics.Short_Complex_Complex_Types
  * Float

    Ada.Numerics.Complex_Complex_Types
  * Long_Float

    Ada.Numerics.Long_Complex_Complex_Types

Ada.Numerics.Generic_Elementary_Functions
: This is a generic package that provides the implementation of standard elementary functions (such as log an trigonometric functions) for an arbitrary float type.

  The following predefined instantiations of this package exist

  * Short_Float

    Ada.Numerics.Short_Elementary_Functions

* Float

  Ada.Numerics.Elementary_Functions
* Long_Float

  Ada.Numerics.Long_Elementary_Functions

Ada.Numerics.Generic_Real_Arrays '(G.3.1)'
: Generic operations on arrays of reals

Ada.Numerics.Real_Arrays '(G.3.1)'
: Preinstantiation of Ada.Numerics.Generic_Real_Arrays (Float).

Ada.Real_Time '(D.8)'
: This package provides facilities similar to those of Calendar, but operating with a finer clock suitable for real time control. Note that annex D requires that there be no backward clock jumps, and GNAT generally guarantees this behavior, but of course if the external clock on which the GNAT runtime depends is deliberately reset by some external event, then such a backward jump may occur.

Ada.Real_Time.Timing_Events '(D.15)'
: This package allows procedures to be executed at a specified time without the use of a task or a delay statement.

Ada.Sequential_IO '(A.8.1)'
: This package provides input-output facilities for sequential files, which can contain a sequence of values of a single type, which can be any Ada type, including indefinite (unconstrained) types.

Ada.Storage_IO '(A.9)'
: This package provides a facility for mapping arbitrary Ada types to and from a storage buffer. It is primarily intended for the creation of new IO packages.

Ada.Streams '(13.13.1)'
: This is a generic package that provides the basic support for the concept of streams as used by the stream attributes (Input, Output, Read and Write).

Ada.Streams.Stream_IO '(A.12.1)'
: This package is a specialization of the type Streams defined in package Streams together with a set of operations providing Stream_IO capability. The Stream_IO model permits both random and sequential access to a file which can contain an arbitrary set of values of one or more Ada types.

Ada.Strings '(A.4.1)'
: This package provides some basic constants used by the string handling packages.

Ada.Strings.Bounded '(A.4.4)'
: This package provides facilities for handling variable length strings. The bounded model requires a maximum length. It is thus somewhat more limited than the unbounded model, but avoids the use of dynamic allocation or finalization.

`Ada.Strings.Bounded.Equal_Case_Insensitive` '(A.4.10)'
: Provides case-insensitive comparisons of bounded strings

`Ada.Strings.Bounded.Hash` '(A.4.9)'
: This package provides a generic hash function for bounded strings

`Ada.Strings.Bounded.Hash_Case_Insensitive` '(A.4.9)'
: This package provides a generic hash function for bounded strings that converts the string to be hashed to lower case.

`Ada.Strings.Bounded.Less_Case_Insensitive` '(A.4.10)'
: This package provides a comparison function for bounded strings that works in a case insensitive manner by converting to lower case before the comparison.

`Ada.Strings.Fixed` '(A.4.3)'
: This package provides facilities for handling fixed length strings.

`Ada.Strings.Fixed.Equal_Case_Insensitive` '(A.4.10)'
: This package provides an equality function for fixed strings that compares the strings after converting both to lower case.

`Ada.Strings.Fixed.Hash_Case_Insensitive` '(A.4.9)'
: This package provides a case insensitive hash function for fixed strings that converts the string to lower case before computing the hash.

`Ada.Strings.Fixed.Less_Case_Insensitive` '(A.4.10)'
: This package provides a comparison function for fixed strings that works in a case insensitive manner by converting to lower case before the comparison.

`Ada.Strings.Hash` '(A.4.9)'
: This package provides a hash function for strings.

`Ada.Strings.Hash_Case_Insensitive` '(A.4.9)'
: This package provides a hash function for strings that is case insensitive. The string is converted to lower case before computing the hash.

`Ada.Strings.Less_Case_Insensitive` '(A.4.10)'
: This package provides a comparison function for\strings that works in a case insensitive manner by converting to lower case before the comparison.

`Ada.Strings.Maps` '(A.4.2)'
: This package provides facilities for handling character mappings and arbitrarily defined subsets of characters. For instance it is useful in defining specialized translation tables.

`Ada.Strings.Maps.Constants` '(A.4.6)'
: This package provides a standard set of predefined mappings and predefined character sets. For example, the standard upper to lower case conversion table is found in this package. Note that upper to lower case conversion is non-trivial if you want to take the entire set of characters, including extended characters like E with an acute accent, into account. You should use the mappings in this package (rather than adding 32 yourself) to do case mappings.

`Ada.Strings.Unbounded` '(A.4.5)'
: This package provides facilities for handling variable length strings. The unbounded model allows arbitrary length strings, but requires the use of dynamic allocation and finalization.

`Ada.Strings.Unbounded.Equal_Case_Insensitive` '(A.4.10)'
: Provides case-insensitive comparisons of unbounded strings

`Ada.Strings.Unbounded.Hash` '(A.4.9)'
: This package provides a generic hash function for unbounded strings

`Ada.Strings.Unbounded.Hash_Case_Insensitive` '(A.4.9)'
: This package provides a generic hash function for unbounded strings that converts the string to be hashed to lower case.

`Ada.Strings.Unbounded.Less_Case_Insensitive` '(A.4.10)'
: This package provides a comparison function for unbounded strings that works in a case insensitive manner by converting to lower case before the comparison.

`Ada.Strings.UTF_Encoding` '(A.4.11)'
: This package provides basic definitions for dealing with UTF-encoded strings.

`Ada.Strings.UTF_Encoding.Conversions` '(A.4.11)'
: This package provides conversion functions for UTF-encoded strings.

`Ada.Strings.UTF_Encoding.Strings` '(A.4.11)'

`Ada.Strings.UTF_Encoding.Wide_Strings` '(A.4.11)'

`Ada.Strings.UTF_Encoding.Wide_Wide_Strings` '(A.4.11)'
: These packages provide facilities for handling UTF encodings for Strings, Wide_Strings and Wide_Wide_Strings.

`Ada.Strings.Wide_Bounded` '(A.4.7)'

`Ada.Strings.Wide_Fixed` '(A.4.7)'

`Ada.Strings.Wide_Maps` '(A.4.7)'

`Ada.Strings.Wide_Unbounded` '(A.4.7)'
: These packages provide analogous capabilities to the corresponding packages without `Wide_` in the name, but operate with the types `Wide_String` and `Wide_Character` instead of `String` and `Character`. Versions of all the child packages are available.

`Ada.Strings.Wide_Wide_Bounded` '(A.4.7)'

`Ada.Strings.Wide_Wide_Fixed` '(A.4.7)'

`Ada.Strings.Wide_Wide_Maps` '(A.4.7)'

`Ada.Strings.Wide_Wide_Unbounded` '(A.4.7)'
: These packages provide analogous capabilities to the corresponding packages without `Wide_` in the name, but operate with the types `Wide_Wide_String` and `Wide_Wide_Character` instead of `String` and `Character`.

`Ada.Synchronous_Barriers` '(D.10.1)'
: This package provides facilities for synchronizing tasks at a low level with barriers.

`Ada.Synchronous_Task_Control` '(D.10)'
: This package provides some standard facilities for controlling task communication in a synchronous manner.

`Ada.Synchronous_Task_Control.EDF` '(D.10)'
: Not implemented in GNAT.

`Ada.Tags`
: This package contains definitions for manipulation of the tags of tagged values.

`Ada.Tags.Generic_Dispatching_Constructor` '(3.9)'
: This package provides a way of constructing tagged class-wide values given only the tag value.

`Ada.Task_Attributes` '(C.7.2)'
: This package provides the capability of associating arbitrary task-specific data with separate tasks.

`Ada.Task_Identification` '(C.7.1)'
: This package provides capabilities for task identification.

`Ada.Task_Termination` '(C.7.3)'
: This package provides control over task termination.

`Ada.Text_IO`
: This package provides basic text input-output capabilities for character, string and numeric data. The subpackages of this package are listed next. Note that although these are defined as subpackages in the RM, they are actually transparently implemented as child packages in GNAT, meaning that they are only loaded if needed.

`Ada.Text_IO.Decimal_IO`
: Provides input-output facilities for decimal fixed-point types

`Ada.Text_IO.Enumeration_IO`
: Provides input-output facilities for enumeration types.

`Ada.Text_IO.Fixed_IO`
: Provides input-output facilities for ordinary fixed-point types.

`Ada.Text_IO.Float_IO`
: Provides input-output facilities for float types. The following predefined instantiations of this generic package are available:

  * `Short_Float`

    `Short_Float_Text_IO`

  * `Float`

    `Float_Text_IO`

  * `Long_Float`

    `Long_Float_Text_IO`

`Ada.Text_IO.Integer_IO`
: Provides input-output facilities for integer types. The following predefined instantiations of this generic package are available:

  * `Short_Short_Integer`
    `Ada.Short_Short_Integer_Text_IO`
  * `Short_Integer`
    `Ada.Short_Integer_Text_IO`
  * `Integer`
    `Ada.Integer_Text_IO`
  * `Long_Integer`
    `Ada.Long_Integer_Text_IO`
  * `Long_Long_Integer`
    `Ada.Long_Long_Integer_Text_IO`

`Ada.Text_IO.Modular_IO`
: Provides input-output facilities for modular (unsigned) types.

`Ada.Text_IO.Bounded_IO (A.10.11)`
: Provides input-output facilities for bounded strings.

`Ada.Text_IO.Complex_IO (G.1.3)`
: This package provides basic text input-output capabilities for complex data.

`Ada.Text_IO.Editing (F.3.3)`
: This package contains routines for edited output, analogous to the use of pictures in COBOL. The picture formats used by this package are a close copy of the facility in COBOL.

`Ada.Text_IO.Text_Streams (A.12.2)`
: This package provides a facility that allows Text_IO files to be treated as streams, so that the stream attributes can be used for writing arbitrary data, including binary data, to Text_IO files.

`Ada.Text_IO.Unbounded_IO (A.10.12)`
: This package provides input-output facilities for unbounded strings.

`Ada.Unchecked_Conversion (13.9)`
: This generic package allows arbitrary conversion from one type to another of the same size, providing for breaking the type safety in special circumstances.

  If the types have the same Size (more accurately the same Value_Size), then the effect is simply to transfer the bits from the source to the target type without any modification. This usage is well defined, and for simple types whose representation is typically the same across all implementations, gives a portable method of performing such conversions.

  If the types do not have the same size, then the result is implementation defined, and thus may be non-portable. The following describes how GNAT handles such unchecked conversion cases.

  If the types are of different sizes, and are both discrete types, then the effect is of a normal type conversion without any constraint checking. In particular if the result type has a larger size, the result will be zero or sign extended. If the result type has a smaller size, the result will be truncated by ignoring high order bits.

If the types are of different sizes, and are not both discrete types, then the conversion works as though pointers were created to the source and target, and the pointer value is converted. The effect is that bits are copied from successive low order storage units and bits of the source up to the length of the target type.

A warning is issued if the lengths differ, since the effect in this case is implementation dependent, and the above behavior may not match that of some other compiler.

A pointer to one type may be converted to a pointer to another type using unchecked conversion. The only case in which the effect is undefined is when one or both pointers are pointers to unconstrained array types. In this case, the bounds information may get incorrectly transferred, and in particular, GNAT uses double size pointers for such types, and it is meaningless to convert between such pointer types. GNAT will issue a warning if the alignment of the target designated type is more strict than the alignment of the source designated type (since the result may be unaligned in this case).

A pointer other than a pointer to an unconstrained array type may be converted to and from System.Address. Such usage is common in Ada 83 programs, but note that Ada.Address_To_Access_Conversions is the preferred method of performing such conversions in Ada 95 and Ada 2005. Neither unchecked conversion nor Ada.Address_To_Access_Conversions should be used in conjunction with pointers to unconstrained objects, since the bounds information cannot be handled correctly in this case.

`Ada.Unchecked_Deallocation` '(13.11.2)'
: This generic package allows explicit freeing of storage previously allocated by use of an allocator.

`Ada.Wide_Text_IO` '(A.11)'
: This package is similar to `Ada.Text_IO`, except that the external file supports wide character representations, and the internal types are `Wide_Character` and `Wide_String` instead of `Character` and `String`. The corresponding set of nested packages and child packages are defined.

`Ada.Wide_Wide_Text_IO` '(A.11)'
: This package is similar to `Ada.Text_IO`, except that the external file supports wide character representations, and the internal types are `Wide_Character` and `Wide_String` instead of `Character` and `String`. The corresponding set of nested packages and child packages are defined.

For packages in Interfaces and System, all the RM defined packages are available in GNAT, see the Ada 2012 RM for full details.

11 The Implementation of Standard I/O

GNAT implements all the required input-output facilities described in A.6 through A.14. These sections of the Ada Reference Manual describe the required behavior of these packages from the Ada point of view, and if you are writing a portable Ada program that does not need to know the exact manner in which Ada maps to the outside world when it comes to reading or writing external files, then you do not need to read this chapter. As long as your files are all regular files (not pipes or devices), and as long as you write and read the files only from Ada, the description in the Ada Reference Manual is sufficient.

However, if you want to do input-output to pipes or other devices, such as the keyboard or screen, or if the files you are dealing with are either generated by some other language, or to be read by some other language, then you need to know more about the details of how the GNAT implementation of these input-output facilities behaves.

In this chapter we give a detailed description of exactly how GNAT interfaces to the file system. As always, the sources of the system are available to you for answering questions at an even more detailed level, but for most purposes the information in this chapter will suffice.

Another reason that you may need to know more about how input-output is implemented arises when you have a program written in mixed languages where, for example, files are shared between the C and Ada sections of the same program. GNAT provides some additional facilities, in the form of additional child library packages, that facilitate this sharing, and these additional facilities are also described in this chapter.

11.1 Standard I/O Packages

The Standard I/O packages described in Annex A for

* Ada.Text_IO
* Ada.Text_IO.Complex_IO
* Ada.Text_IO.Text_Streams
* Ada.Wide_Text_IO
* Ada.Wide_Text_IO.Complex_IO
* Ada.Wide_Text_IO.Text_Streams
* Ada.Wide_Wide_Text_IO
* Ada.Wide_Wide_Text_IO.Complex_IO
* Ada.Wide_Wide_Text_IO.Text_Streams
* Ada.Stream_IO
* Ada.Sequential_IO
* Ada.Direct_IO

are implemented using the C library streams facility; where

* All files are opened using `fopen`.
* All input/output operations use `fread`/`fwrite`.

There is no internal buffering of any kind at the Ada library level. The only buffering is that provided at the system level in the implementation of the library routines that support streams. This facilitates shared use of these streams by mixed language programs. Note though that system level buffering is explicitly enabled at elaboration of the standard I/O packages and that can have an impact on mixed language programs, in particular those using I/O before calling the Ada elaboration routine (e.g., adainit). It is recommended to call the Ada elaboration routine before performing any I/O or when impractical, flush the common I/O streams and in particular Standard_Output before elaborating the Ada code.

11.2 FORM Strings

The format of a FORM string in GNAT is:

```
"keyword=value,keyword=value,...,keyword=value"
```

where letters may be in upper or lower case, and there are no spaces between values. The order of the entries is not important. Currently the following keywords defined.

```
TEXT_TRANSLATION=[YES|NO|TEXT|BINARY|U8TEXT|WTEXT|U16TEXT]
SHARED=[YES|NO]
WCEM=[n|h|u|s|e|8|b]
ENCODING=[UTF8|8BITS]
```

The use of these parameters is described later in this section. If an unrecognized keyword appears in a form string, it is silently ignored and not considered invalid.

11.3 Direct_IO

Direct_IO can only be instantiated for definite types. This is a restriction of the Ada language, which means that the records are fixed length (the length being determined by `type'Size`, rounded up to the next storage unit boundary if necessary).

The records of a Direct_IO file are simply written to the file in index sequence, with the first record starting at offset zero, and subsequent records following. There is no control information of any kind. For example, if 32-bit integers are being written, each record takes 4-bytes, so the record at index `K` starts at offset `(K-1)*4`.

There is no limit on the size of Direct_IO files, they are expanded as necessary to accommodate whatever records are written to the file.

11.4 Sequential_IO

Sequential_IO may be instantiated with either a definite (constrained) or indefinite (unconstrained) type.

For the definite type case, the elements written to the file are simply the memory images of the data values with no control information of any kind. The resulting file should be read using the same type, no validity checking is performed on input.

For the indefinite type case, the elements written consist of two parts. First is the size of the data item, written as the memory image of a `Interfaces.C.size_t` value, followed by the memory image of the data value. The resulting file can only be read using the same (unconstrained) type. Normal assignment checks are performed on these read operations, and if these checks fail, `Data_Error` is raised. In particular, in the array case, the lengths must match, and in the variant record case, if the variable for a particular read operation is constrained, the discriminants must match.

Note that it is not possible to use Sequential_IO to write variable length array items, and then read the data back into different length arrays. For example, the following will raise `Data_Error`:

```
package IO is new Sequential_IO (String);
F : IO.File_Type;
S : String (1..4);
...
IO.Create (F)
IO.Write (F, "hello!")
IO.Reset (F, Mode=>In_File);
IO.Read (F, S);
Put_Line (S);
```

On some Ada implementations, this will print `hell`, but the program is clearly incorrect, since there is only one element in the file, and that element is the string `hello!`.

In Ada 95 and Ada 2005, this kind of behavior can be legitimately achieved using Stream_IO, and this is the preferred mechanism. In particular, the above program fragment rewritten to use Stream_IO will work correctly.

11.5 Text_IO

Text_IO files consist of a stream of characters containing the following special control characters:

```
LF (line feed, 16#0A#) Line Mark
FF (form feed, 16#0C#) Page Mark
```

A canonical Text_IO file is defined as one in which the following conditions are met:

* The character `LF` is used only as a line mark, i.e., to mark the end of the line.
* The character `FF` is used only as a page mark, i.e., to mark the end of a page and consequently can appear only immediately following a `LF` (line mark) character.
* The file ends with either `LF` (line mark) or `LF`-*FF* (line mark, page mark). In the former case, the page mark is implicitly assumed to be present.

A file written using Text_IO will be in canonical form provided that no explicit `LF` or `FF` characters are written using `Put` or `Put_Line`. There will be no `FF` character at the end of the file unless an explicit `New_Page` operation was performed before closing the file.

A canonical Text_IO file that is a regular file (i.e., not a device or a pipe) can be read using any of the routines in Text_IO. The semantics in this case will be exactly as defined in the Ada Reference Manual, and all the routines in Text_IO are fully implemented.

A text file that does not meet the requirements for a canonical Text_IO file has one of the following:

* The file contains `FF` characters not immediately following a `LF` character.
* The file contains `LF` or `FF` characters written by `Put` or `Put_Line`, which are not logically considered to be line marks or page marks.

* The file ends in a character other than LF or FF, i.e., there is no explicit line mark or page mark at the end of the file.

Text_IO can be used to read such non-standard text files but subprograms to do with line or page numbers do not have defined meanings. In particular, a FF character that does not follow a LF character may or may not be treated as a page mark from the point of view of page and line numbering. Every LF character is considered to end a line, and there is an implied LF character at the end of the file.

11.5.1 Stream Pointer Positioning

Ada.Text_IO has a definition of current position for a file that is being read. No internal buffering occurs in Text_IO, and usually the physical position in the stream used to implement the file corresponds to this logical position defined by Text_IO. There are two exceptions:

* After a call to End_Of_Page that returns True, the stream is positioned past the LF (line mark) that precedes the page mark. Text_IO maintains an internal flag so that subsequent read operations properly handle the logical position which is unchanged by the End_Of_Page call.
* After a call to End_Of_File that returns True, if the Text_IO file was positioned before the line mark at the end of file before the call, then the logical position is unchanged, but the stream is physically positioned right at the end of file (past the line mark, and past a possible page mark following the line mark. Again Text_IO maintains internal flags so that subsequent read operations properly handle the logical position.

These discrepancies have no effect on the observable behavior of Text_IO, but if a single Ada stream is shared between a C program and Ada program, or shared (using shared=yes in the form string) between two Ada files, then the difference may be observable in some situations.

11.5.2 Reading and Writing Non-Regular Files

A non-regular file is a device (such as a keyboard), or a pipe. Text_IO can be used for reading and writing. Writing is not affected and the sequence of characters output is identical to the normal file case, but for reading, the behavior of Text_IO is modified to avoid undesirable look-ahead as follows:

An input file that is not a regular file is considered to have no page marks. Any Ascii.FF characters (the character normally used for a page mark) appearing in the file are considered to be data characters. In particular:

* Get_Line and Skip_Line do not test for a page mark following a line mark. If a page mark appears, it will be treated as a data character.
* This avoids the need to wait for an extra character to be typed or entered from the pipe to complete one of these operations.
* End_Of_Page always returns False
* End_Of_File will return False if there is a page mark at the end of the file.

Output to non-regular files is the same as for regular files. Page marks may be written to non-regular files using New_Page, but as noted above they will not be treated as page marks on input if the output is piped to another Ada program.

Another important discrepancy when reading non-regular files is that the end of file indication is not 'sticky'. If an end of file is entered, e.g., by pressing the `EOT` key, then end of file is signaled once (i.e., the test `End_Of_File` will yield `True`, or a read will raise `End_Error`), but then reading can resume to read data past that end of file indication, until another end of file indication is entered.

11.5.3 Get_Immediate

Get_Immediate returns the next character (including control characters) from the input file. In particular, Get_Immediate will return LF or FF characters used as line marks or page marks. Such operations leave the file positioned past the control character, and it is thus not treated as having its normal function. This means that page, line and column counts after this kind of Get_Immediate call are set as though the mark did not occur. In the case where a Get_Immediate leaves the file positioned between the line mark and page mark (which is not normally possible), it is undefined whether the FF character will be treated as a page mark.

11.5.4 Treating Text_IO Files as Streams

The package `Text_IO.Streams` allows a `Text_IO` file to be treated as a stream. Data written to a `Text_IO` file in this stream mode is binary data. If this binary data contains bytes 16#0A# (`LF`) or 16#0C# (`FF`), the resulting file may have non-standard format. Similarly if read operations are used to read from a Text_IO file treated as a stream, then `LF` and `FF` characters may be skipped and the effect is similar to that described above for `Get_Immediate`.

11.5.5 Text_IO Extensions

A package GNAT.IO_Aux in the GNAT library provides some useful extensions to the standard `Text_IO` package:

* function File_Exists (Name : String) return Boolean; Determines if a file of the given name exists.
* function Get_Line return String; Reads a string from the standard input file. The value returned is exactly the length of the line that was read.
* function Get_Line (File : Ada.Text_IO.File_Type) return String; Similar, except that the parameter File specifies the file from which the string is to be read.

11.5.6 Text_IO Facilities for Unbounded Strings

The package `Ada.Strings.Unbounded.Text_IO` in library files `a-suteio.ads/adb` contains some GNAT-specific subprograms useful for Text_IO operations on unbounded strings:

* function Get_Line (File : File_Type) return Unbounded_String; Reads a line from the specified file and returns the result as an unbounded string.
* procedure Put (File : File_Type; U : Unbounded_String); Writes the value of the given unbounded string to the specified file Similar to the effect of `Put (To_String (U))` except that an extra copy is avoided.
* procedure Put_Line (File : File_Type; U : Unbounded_String); Writes the value of the given unbounded string to the specified file, followed by a `New_Line`. Similar to the effect of `Put_Line (To_String (U))` except that an extra copy is avoided.

In the above procedures, `File` is of type `Ada.Text_IO.File_Type` and is optional. If the parameter is omitted, then the standard input or output file is referenced as appropriate.

The package `Ada.Strings.Wide_Unbounded.Wide_Text_IO` in library files `a-swuwti.ads` and `a-swuwti.adb` provides similar extended `Wide_Text_IO` functionality for unbounded wide strings.

The package `Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Text_IO` in library files `a-szuzti.ads` and `a-szuzti.adb` provides similar extended `Wide_Wide_Text_IO` functionality for unbounded wide wide strings.

11.6 Wide_Text_IO

`Wide_Text_IO` is similar in most respects to Text_IO, except that both input and output files may contain special sequences that represent wide character values. The encoding scheme for a given file may be specified using a FORM parameter:

```
WCEM=`x`
```

as part of the FORM string (WCEM = wide character encoding method), where `x` is one of the following characters

Character	**Encoding**
'h'	Hex ESC encoding
'u'	Upper half encoding
's'	Shift-JIS encoding
'e'	EUC Encoding
'8'	UTF-8 encoding
'b'	Brackets encoding

The encoding methods match those that can be used in a source program, but there is no requirement that the encoding method used for the source program be the same as the encoding method used for files, and different files may use different encoding methods.

The default encoding method for the standard files, and for opened files for which no WCEM parameter is given in the FORM string matches the wide character encoding specified for the main program (the default being brackets encoding if no coding method was specified with -gnatW).

'Hex Coding'

In this encoding, a wide character is represented by a five character sequence:

```
ESC a b c d
```

where `a`, `b`, `c`, `d` are the four hexadecimal characters (using upper case letters) of the wide character code. For example, ESC A345 is used to represent the wide character with code 16#A345#. This scheme is compatible with use of the full `Wide_Character` set.

'Upper Half Coding'
: The wide character with encoding 16#abcd#, where the upper bit is on (i.e., a is in the range 8-F) is represented as two bytes 16#ab# and 16#cd#. The second byte may never be a format control character, but is not required to be in the upper half. This method can be also used for shift-JIS or EUC where the internal coding matches the external coding.

'Shift JIS Coding'
: A wide character is represented by a two character sequence 16#ab# and 16#cd#, with the restrictions described for upper half encoding as described above. The internal character code is the corresponding JIS character according to the standard algorithm for Shift-JIS conversion. Only characters defined in the JIS code set table can be used with this encoding method.

'EUC Coding'
: A wide character is represented by a two character sequence 16#ab# and 16#cd#, with both characters being in the upper half. The internal character code is the corresponding JIS character according to the EUC encoding algorithm. Only characters defined in the JIS code set table can be used with this encoding method.

'UTF-8 Coding'
: A wide character is represented using UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO 10646-1/Am.2. Depending on the character value, the representation is a one, two, or three byte sequence:

```
16#0000#-16#007f#: 2#0xxxxxxx#
16#0080#-16#07ff#: 2#110xxxxx# 2#10xxxxxx#
16#0800#-16#ffff#: 2#1110xxxx# 2#10xxxxxx# 2#10xxxxxx#
```

where the `xxx` bits correspond to the left-padded bits of the 16-bit character value. Note that all lower half ASCII characters are represented as ASCII bytes and all upper half characters and other wide characters are represented as sequences of upper-half (The full UTF-8 scheme allows for encoding 31-bit characters as 6-byte sequences, but in this implementation, all UTF-8 sequences of four or more bytes length will raise a Constraint_Error, as will all invalid UTF-8 sequences.)

'Brackets Coding'
: In this encoding, a wide character is represented by the following eight character sequence:

```
[ " a b c d " ]
```

where `a`, `b`, `c`, `d` are the four hexadecimal characters (using uppercase letters) of the wide character code. For example, `["A345"]` is used to represent the wide character with code `16#A345#`. This scheme is compatible with use of the full Wide_Character set. On input, brackets coding can also be used for upper half characters, e.g., `["C1"]` for lower case a.

However, on output, brackets notation is only used for wide characters with a code greater than `16#FF#`.

Note that brackets coding is not normally used in the context of Wide_Text_IO or Wide_Wide_Text_IO, since it is really just designed as a portable way of encoding source files. In the context of Wide_Text_IO or Wide_Wide_Text_IO, it can only be used if the file does not contain any instance of the left bracket character other than to encode wide character values using the brackets encoding method. In practice it is expected that some standard wide character encoding method such as UTF-8 will be used for text input output.

If brackets notation is used, then any occurrence of a left bracket in the input file which is not the start of a valid wide character sequence will cause Constraint_Error to be raised. It is possible to encode a left bracket as [“5B”] and Wide_Text_IO and Wide_Wide_Text_IO input will interpret this as a left bracket.

However, when a left bracket is output, it will be output as a left bracket and not as [“5B”]. We make this decision because for normal use of Wide_Text_IO for outputting messages, it is unpleasant to clobber left brackets. For example, if we write:

```
Put_Line ("Start of output [first run]");
```

we really do not want to have the left bracket in this message clobbered so that the output reads:

```
Start of output ["5B"]first run]
```

In practice brackets encoding is reasonably useful for normal Put_Line use since we won't get confused between left brackets and wide character sequences in the output. But for input, or when files are written out and read back in, it really makes better sense to use one of the standard encoding methods such as UTF-8.

For the coding schemes other than UTF-8, Hex, or Brackets encoding, not all wide character values can be represented. An attempt to output a character that cannot be represented using the encoding scheme for the file causes Constraint_Error to be raised. An invalid wide character sequence on input also causes Constraint_Error to be raised.

11.6.1 Stream Pointer Positioning

`Ada.Wide_Text_IO` is similar to `Ada.Text_IO` in its handling of stream pointer positioning ([Text_IO], page 246). There is one additional case:

If `Ada.Wide_Text_IO.Look_Ahead` reads a character outside the normal lower ASCII set, i.e. a character in the range:

```
Wide_Character'Val (16#0080#) .. Wide_Character'Val (16#FFFF#)
```

then although the logical position of the file pointer is unchanged by the `Look_Ahead` call, the stream is physically positioned past the wide character sequence. Again this is to avoid the need for buffering or backup, and all `Wide_Text_IO` routines check the internal indication that this situation has occurred so that this is not visible to a normal program using `Wide_Text_IO`. However, this discrepancy can be observed if the wide text file shares a stream with another file.

11.6.2 Reading and Writing Non-Regular Files

As in the case of Text_IO, when a non-regular file is read, it is assumed that the file contains no page marks (any form characters are treated as data characters), and `End_Of_Page` always returns `False`. Similarly, the end of file indication is not sticky, so it is possible to read beyond an end of file.

11.7 Wide_Wide_Text_IO

`Wide_Wide_Text_IO` is similar in most respects to Text_IO, except that both input and output files may contain special sequences that represent wide wide character values. The encoding scheme for a given file may be specified using a FORM parameter:

```
WCEM=`x`
```

as part of the FORM string (WCEM = wide character encoding method), where `x` is one of the following characters

Character	**Encoding**
'h'	Hex ESC encoding
'u'	Upper half encoding
's'	Shift-JIS encoding
'e'	EUC Encoding
'8'	UTF-8 encoding
'b'	Brackets encoding

The encoding methods match those that can be used in a source program, but there is no requirement that the encoding method used for the source program be the same as the encoding method used for files, and different files may use different encoding methods.

The default encoding method for the standard files, and for opened files for which no WCEM parameter is given in the FORM string matches the wide character encoding specified for the main program (the default being brackets encoding if no coding method was specified with -gnatW).

'UTF-8 Coding'

A wide character is represented using UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO 10646-1/Am.2. Depending on the character value, the representation is a one, two, three, or four byte sequence:

```
16#000000#-16#00007f#: 2#0xxxxxxx#
16#000080#-16#0007ff#: 2#110xxxxx# 2#10xxxxxx#
16#000800#-16#00ffff#: 2#1110xxxx# 2#10xxxxxx# 2#10xxxxxx#
16#010000#-16#10ffff#: 2#11110xxx# 2#10xxxxxx# 2#10xxxxxx# 2#10xxxxxx#
```

where the `xxx` bits correspond to the left-padded bits of the 21-bit character value. Note that all lower half ASCII characters are represented as

ASCII bytes and all upper half characters and other wide characters are represented as sequences of upper-half characters.

'Brackets Coding'

In this encoding, a wide wide character is represented by the following eight character sequence if is in wide character range

```
[ " a b c d " ]
```

and by the following ten character sequence if not

```
[ " a b c d e f " ]
```

where `a`, `b`, `c`, `d`, `e`, and `f` are the four or six hexadecimal characters (using uppercase letters) of the wide wide character code. For example, `["01A345"]` is used to represent the wide wide character with code `16#01A345#`.

This scheme is compatible with use of the full Wide_Wide_Character set. On input, brackets coding can also be used for upper half characters, e.g., `["C1"]` for lower case a. However, on output, brackets notation is only used for wide characters with a code greater than `16#FF#`.

If is also possible to use the other Wide_Character encoding methods, such as Shift-JIS, but the other schemes cannot support the full range of wide wide characters. An attempt to output a character that cannot be represented using the encoding scheme for the file causes Constraint_Error to be raised. An invalid wide character sequence on input also causes Constraint_Error to be raised.

11.7.1 Stream Pointer Positioning

`Ada.Wide_Wide_Text_IO` is similar to `Ada.Text_IO` in its handling of stream pointer positioning ([Text_IO], page 246). There is one additional case:

If `Ada.Wide_Wide_Text_IO.Look_Ahead` reads a character outside the normal lower ASCII set, i.e. a character in the range:

```
Wide_Wide_Character'Val (16#0080#) .. Wide_Wide_Character'Val (16#10FFFF#)
```

then although the logical position of the file pointer is unchanged by the `Look_Ahead` call, the stream is physically positioned past the wide character sequence. Again this is to avoid the need for buffering or backup, and all `Wide_Wide_Text_IO` routines check the internal indication that this situation has occurred so that this is not visible to a normal program using `Wide_Wide_Text_IO`. However, this discrepancy can be observed if the wide text file shares a stream with another file.

11.7.2 Reading and Writing Non-Regular Files

As in the case of Text_IO, when a non-regular file is read, it is assumed that the file contains no page marks (any form characters are treated as data characters), and `End_Of_Page` always returns `False`. Similarly, the end of file indication is not sticky, so it is possible to read beyond an end of file.

11.8 Stream_IO

A stream file is a sequence of bytes, where individual elements are written to the file as described in the Ada Reference Manual. The type `Stream_Element` is simply a byte. There are two ways to read or write a stream file.

* The operations `Read` and `Write` directly read or write a sequence of stream elements with no control information.
* The stream attributes applied to a stream file transfer data in the manner described for stream attributes.

11.9 Text Translation

`Text_Translation=xxx` may be used as the Form parameter passed to Text_IO.Create and Text_IO.Open. `Text_Translation=xxx` has no effect on Unix systems. Possible values are:

* `Yes` or `Text` is the default, which means to translate LF to/from CR/LF on Windows systems.

  `No` disables this translation; i.e. it uses binary mode. For output files, `Text_Translation=No` may be used to create Unix-style files on Windows.
* `wtext` translation enabled in Unicode mode. (corresponds to _O_WTEXT).
* `u8text` translation enabled in Unicode UTF-8 mode. (corresponds to O_U8TEXT).
* `u16text` translation enabled in Unicode UTF-16 mode. (corresponds to_O_U16TEXT).

11.10 Shared Files

Section A.14 of the Ada Reference Manual allows implementations to provide a wide variety of behavior if an attempt is made to access the same external file with two or more internal files.

To provide a full range of functionality, while at the same time minimizing the problems of portability caused by this implementation dependence, GNAT handles file sharing as follows:

* In the absence of a `shared=xxx` form parameter, an attempt to open two or more files with the same full name is considered an error and is not supported. The exception `Use_Error` will be raised. Note that a file that is not explicitly closed by the program remains open until the program terminates.
* If the form parameter `shared=no` appears in the form string, the file can be opened or created with its own separate stream identifier, regardless of whether other files sharing the same external file are opened. The exact effect depends on how the C stream routines handle multiple accesses to the same external files using separate streams.
* If the form parameter `shared=yes` appears in the form string for each of two or more files opened using the same full name, the same stream is shared between these files, and the semantics are as described in Ada Reference Manual, Section A.14.

When a program that opens multiple files with the same name is ported from another Ada compiler to GNAT, the effect will be that `Use_Error` is raised.

The documentation of the original compiler and the documentation of the program should then be examined to determine if file sharing was expected, and `shared=xxx` parameters added to `Open` and `Create` calls as required.

When a program is ported from GNAT to some other Ada compiler, no special attention is required unless the `shared=xxx` form parameter is used in the program. In this case, you must examine the documentation of the new compiler to see if it supports the required file sharing semantics, and form strings modified appropriately. Of course it may be the case that the program cannot be ported if the target compiler does not support the required functionality. The best approach in writing portable code is to avoid file sharing (and hence the use of the `shared=xxx` parameter in the form string) completely.

One common use of file sharing in Ada 83 is the use of instantiations of Sequential_IO on the same file with different types, to achieve heterogeneous input-output. Although this approach will work in GNAT if `shared=yes` is specified, it is preferable in Ada to use Stream_IO for this purpose (using the stream attributes).

11.11 Filenames encoding

An encoding form parameter can be used to specify the filename encoding `encoding=xxx`.

* If the form parameter `encoding=utf8` appears in the form string, the filename must be encoded in UTF-8.
* If the form parameter `encoding=8bits` appears in the form string, the filename must be a standard 8bits string.

In the absence of a `encoding=xxx` form parameter, the encoding is controlled by the `GNAT_CODE_PAGE` environment variable. And if not set `utf8` is assumed.

'CP_ACP'

The current system Windows ANSI code page.

'CP_UTF8'

UTF-8 encoding

This encoding form parameter is only supported on the Windows platform. On the other Operating Systems the run-time is supporting UTF-8 natively.

11.12 File content encoding

For text files it is possible to specify the encoding to use. This is controlled by the by the `GNAT_CCS_ENCODING` environment variable. And if not set `TEXT` is assumed.

The possible values are those supported on Windows:

'TEXT'

Translated text mode

'WTEXT'

Translated unicode encoding

'U16TEXT'

Unicode 16-bit encoding

'U8TEXT'

Unicode 8-bit encoding

This encoding is only supported on the Windows platform.

11.13 Open Modes

`Open` and `Create` calls result in a call to `fopen` using the mode shown in the following table:

`Open` and `Create` Call Modes

	'OPEN'	'CREATE'
Append_File	"r+"	"w+"
In_File	"r"	"w+"
Out_File (Direct_IO)	"r+"	"w"
Out_File (all other cases)	"w"	"w"
Inout_File	"r+"	"w+"

If text file translation is required, then either `b` or `t` is added to the mode, depending on the setting of Text. Text file translation refers to the mapping of CR/LF sequences in an external file to LF characters internally. This mapping only occurs in DOS and DOS-like systems, and is not relevant to other systems.

A special case occurs with Stream_IO. As shown in the above table, the file is initially opened in `r` or `w` mode for the `In_File` and `Out_File` cases. If a `Set_Mode` operation subsequently requires switching from reading to writing or vice-versa, then the file is reopened in `r+` mode to permit the required operation.

11.14 Operations on C Streams

The package `Interfaces.C_Streams` provides an Ada program with direct access to the C library functions for operations on C streams:

```
package Interfaces.C_Streams is
  -- Note: the reason we do not use the types that are in
  -- Interfaces.C is that we want to avoid dragging in the
  -- code in this unit if possible.
  subtype chars is System.Address;
  -- Pointer to null-terminated array of characters
  subtype FILEs is System.Address;
  -- Corresponds to the C type FILE*
  subtype voids is System.Address;
  -- Corresponds to the C type void*
  subtype int is Integer;
  subtype long is Long_Integer;
  -- Note: the above types are subtypes deliberately, and it
  -- is part of this spec that the above correspondences are
  -- guaranteed.  This means that it is legitimate to, for
  -- example, use Integer instead of int.  We provide these
```

```
-- synonyms for clarity, but in some cases it may be
-- convenient to use the underlying types (for example to
-- avoid an unnecessary dependency of a spec on the spec
-- of this unit).
type size_t is mod 2 ** Standard'Address_Size;
NULL_Stream : constant FILEs;
-- Value returned (NULL in C) to indicate an
-- fdopen/fopen/tmpfile error
----------------------------------
-- Constants Defined in stdio.h --
----------------------------------
EOF : constant int;
-- Used by a number of routines to indicate error or
-- end of file
IOFBF : constant int;
IOLBF : constant int;
IONBF : constant int;
-- Used to indicate buffering mode for setvbuf call
SEEK_CUR : constant int;
SEEK_END : constant int;
SEEK_SET : constant int;
-- Used to indicate origin for fseek call
function stdin return FILEs;
function stdout return FILEs;
function stderr return FILEs;
-- Streams associated with standard files
--------------------------
-- Standard C functions --
--------------------------
-- The functions selected below are ones that are
-- available in UNIX (but not necessarily in ANSI C).
-- These are very thin interfaces
-- which copy exactly the C headers.  For more
-- documentation on these functions, see the Microsoft C
-- "Run-Time Library Reference" (Microsoft Press, 1990,
-- ISBN 1-55615-225-6), which includes useful information
-- on system compatibility.
procedure clearerr (stream : FILEs);
function fclose (stream : FILEs) return int;
function fdopen (handle : int; mode : chars) return FILEs;
function feof (stream : FILEs) return int;
function ferror (stream : FILEs) return int;
function fflush (stream : FILEs) return int;
function fgetc (stream : FILEs) return int;
function fgets (strng : chars; n : int; stream : FILEs)
    return chars;
function fileno (stream : FILEs) return int;
```

```
function fopen (filename : chars; Mode : chars)
    return FILEs;
-- Note: to maintain target independence, use
-- text_translation_required, a boolean variable defined in
-- a-sysdep.c to deal with the target dependent text
-- translation requirement.  If this variable is set,
-- then  b/t should be appended to the standard mode
-- argument to set the text translation mode off or on
-- as required.
function fputc (C : int; stream : FILEs) return int;
function fputs (Strng : chars; Stream : FILEs) return int;
function fread
   (buffer : voids;
    size : size_t;
    count : size_t;
    stream : FILEs)
    return size_t;
function freopen
   (filename : chars;
    mode : chars;
    stream : FILEs)
    return FILEs;
function fseek
   (stream : FILEs;
    offset : long;
    origin : int)
    return int;
function ftell (stream : FILEs) return long;
function fwrite
   (buffer : voids;
    size : size_t;
    count : size_t;
    stream : FILEs)
    return size_t;
function isatty (handle : int) return int;
procedure mktemp (template : chars);
-- The return value (which is just a pointer to template)
-- is discarded
procedure rewind (stream : FILEs);
function rmtmp return int;
function setvbuf
   (stream : FILEs;
    buffer : chars;
    mode : int;
    size : size_t)
    return int;
```

```
  function tmpfile return FILEs;
  function ungetc (c : int; stream : FILEs) return int;
  function unlink (filename : chars) return int;
  ---------------------
  -- Extra functions --
  ---------------------
  -- These functions supply slightly thicker bindings than
  -- those above.  They are derived from functions in the
  -- C Run-Time Library, but may do a bit more work than
  -- just directly calling one of the Library functions.
  function is_regular_file (handle : int) return int;
  -- Tests if given handle is for a regular file (result 1)
  -- or for a non-regular file (pipe or device, result 0).
  ---------------------------------
  -- Control of Text/Binary Mode --
  ---------------------------------
  -- If text_translation_required is true, then the following
  -- functions may be used to dynamically switch a file from
  -- binary to text mode or vice versa.  These functions have
  -- no effect if text_translation_required is false (i.e., in
  -- normal UNIX mode).  Use fileno to get a stream handle.
  procedure set_binary_mode (handle : int);
  procedure set_text_mode (handle : int);
  ----------------------------
  -- Full Path Name support --
  ----------------------------
  procedure full_name (nam : chars; buffer : chars);
  -- Given a NUL terminated string representing a file
  -- name, returns in buffer a NUL terminated string
  -- representing the full path name for the file name.
  -- On systems where it is relevant the   drive is also
  -- part of the full path name.  It is the responsibility
  -- of the caller to pass an actual parameter for buffer
  -- that is big enough for any full path name.  Use
  -- max_path_len given below as the size of buffer.
  max_path_len : integer;
  -- Maximum length of an allowable full path name on the
  -- system, including a terminating NUL character.
end Interfaces.C_Streams;
```

11.15 Interfacing to C Streams

The packages in this section permit interfacing Ada files to C Stream operations.

```
with Interfaces.C_Streams;
package Ada.Sequential_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
```

```
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Sequential_IO.C_Streams;

with Interfaces.C_Streams;
package Ada.Direct_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Direct_IO.C_Streams;

with Interfaces.C_Streams;
package Ada.Text_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Text_IO.C_Streams;

with Interfaces.C_Streams;
package Ada.Wide_Text_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Wide_Text_IO.C_Streams;

with Interfaces.C_Streams;
package Ada.Wide_Wide_Text_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
```

```
       C_Stream : in Interfaces.C_Streams.FILEs;
       Form : in String := "");
end Ada.Wide_Wide_Text_IO.C_Streams;

with Interfaces.C_Streams;
package Ada.Stream_IO.C_Streams is
   function C_Stream (F : File_Type)
      return Interfaces.C_Streams.FILEs;
   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      C_Stream : in Interfaces.C_Streams.FILEs;
      Form : in String := "");
end Ada.Stream_IO.C_Streams;
```

In each of these six packages, the `C_Stream` function obtains the `FILE` pointer from a currently opened Ada file. It is then possible to use the `Interfaces.C_Streams` package to operate on this stream, or the stream can be passed to a C program which can operate on it directly. Of course the program is responsible for ensuring that only appropriate sequences of operations are executed.

One particular use of relevance to an Ada program is that the `setvbuf` function can be used to control the buffering of the stream used by an Ada file. In the absence of such a call the standard default buffering is used.

The `Open` procedures in these packages open a file giving an existing C Stream instead of a file name. Typically this stream is imported from a C program, allowing an Ada file to operate on an existing C file.

12 The GNAT Library

The GNAT library contains a number of general and special purpose packages. It represents functionality that the GNAT developers have found useful, and which is made available to GNAT users. The packages described here are fully supported, and upwards compatibility will be maintained in future releases, so you can use these facilities with the confidence that the same functionality will be available in future releases.

The chapter here simply gives a brief summary of the facilities available. The full documentation is found in the spec file for the package. The full sources of these library packages, including both spec and body, are provided with all GNAT releases. For example, to find out the full specifications of the SPITBOL pattern matching capability, including a full tutorial and extensive examples, look in the `g-spipat.ads` file in the library.

For each entry here, the package name (as it would appear in a `with` clause) is given, followed by the name of the corresponding spec file in parentheses. The packages are children in four hierarchies, `Ada`, `Interfaces`, `System`, and `GNAT`, the latter being a GNAT-specific hierarchy.

Note that an application program should only use packages in one of these four hierarchies if the package is defined in the Ada Reference Manual, or is listed in this section of the GNAT Programmers Reference Manual. All other units should be considered internal implementation units and should not be directly `with`ed by application code. The use of a `with` clause that references one of these internal implementation units makes an application potentially dependent on changes in versions of GNAT, and will generate a warning message.

12.1 `Ada.Characters.Latin_9` (`a-chlat9.ads`)

This child of `Ada.Characters` provides a set of definitions corresponding to those in the RM-defined package `Ada.Characters.Latin_1` but with the few modifications required for `Latin-9`. The provision of such a package is specifically authorized by the Ada Reference Manual (RM A.3.3(27)).

12.2 `Ada.Characters.Wide_Latin_1` (`a-cwila1.ads`)

This child of `Ada.Characters` provides a set of definitions corresponding to those in the RM-defined package `Ada.Characters.Latin_1` but with the types of the constants being `Wide_Character` instead of `Character`. The provision of such a package is specifically authorized by the Ada Reference Manual (RM A.3.3(27)).

12.3 `Ada.Characters.Wide_Latin_9` (`a-cwila9.ads`)

This child of `Ada.Characters` provides a set of definitions corresponding to those in the GNAT defined package `Ada.Characters.Latin_9` but with the types of the constants being `Wide_Character` instead of `Character`. The provision of such a package is specifically authorized by the Ada Reference Manual (RM A.3.3(27)).

12.4 `Ada.Characters.Wide_Wide_Latin_1` (`a-chzla1.ads`)

This child of `Ada.Characters` provides a set of definitions corresponding to those in the RM-defined package `Ada.Characters.Latin_1` but with the types of the constants being

Wide_Wide_Character instead of Character. The provision of such a package is specifically authorized by the Ada Reference Manual (RM A.3.3(27)).

12.5 Ada.Characters.Wide_Wide_Latin_9 (a-chzla9.ads)

This child of Ada.Characters provides a set of definitions corresponding to those in the GNAT defined package Ada.Characters.Latin_9 but with the types of the constants being Wide_Wide_Character instead of Character. The provision of such a package is specifically authorized by the Ada Reference Manual (RM A.3.3(27)).

12.6 Ada.Containers.Bounded_Holders (a-coboho.ads)

This child of Ada.Containers defines a modified version of Indefinite_Holders that avoids heap allocation.

12.7 Ada.Command_Line.Environment (a-colien.ads)

This child of Ada.Command_Line provides a mechanism for obtaining environment values on systems where this concept makes sense.

12.8 Ada.Command_Line.Remove (a-colire.ads)

This child of Ada.Command_Line provides a mechanism for logically removing arguments from the argument list. Once removed, an argument is not visible to further calls to the subprograms in Ada.Command_Line. These calls will not see the removed argument.

12.9 Ada.Command_Line.Response_File (a-clrefi.ads)

This child of Ada.Command_Line provides a mechanism facilities for getting command line arguments from a text file, called a "response file". Using a response file allow passing a set of arguments to an executable longer than the maximum allowed by the system on the command line.

12.10 Ada.Direct_IO.C_Streams (a-diocst.ads)

This package provides subprograms that allow interfacing between C streams and Direct_IO. The stream identifier can be extracted from a file opened on the Ada side, and an Ada file can be constructed from a stream opened on the C side.

12.11 Ada.Exceptions.Is_Null_Occurrence (a-einuoc.ads)

This child subprogram provides a way of testing for the null exception occurrence (Null_Occurrence) without raising an exception.

12.12 Ada.Exceptions.Last_Chance_Handler (a-elchha.ads)

This child subprogram is used for handling otherwise unhandled exceptions (hence the name last chance), and perform clean ups before terminating the program. Note that this subprogram never returns.

12.13 `Ada.Exceptions.Traceback` (`a-exctra.ads`)

This child package provides the subprogram (`Tracebacks`) to give a traceback array of addresses based on an exception occurrence.

12.14 `Ada.Sequential_IO.C_Streams` (`a-siocst.ads`)

This package provides subprograms that allow interfacing between C streams and `Sequential_IO`. The stream identifier can be extracted from a file opened on the Ada side, and an Ada file can be constructed from a stream opened on the C side.

12.15 `Ada.Streams.Stream_IO.C_Streams` (`a-ssicst.ads`)

This package provides subprograms that allow interfacing between C streams and `Stream_IO`. The stream identifier can be extracted from a file opened on the Ada side, and an Ada file can be constructed from a stream opened on the C side.

12.16 `Ada.Strings.Unbounded.Text_IO` (`a-suteio.ads`)

This package provides subprograms for Text_IO for unbounded strings, avoiding the necessity for an intermediate operation with ordinary strings.

12.17 `Ada.Strings.Wide_Unbounded.Wide_Text_IO` (`a-swuwti.ads`)

This package provides subprograms for Text_IO for unbounded wide strings, avoiding the necessity for an intermediate operation with ordinary wide strings.

12.18 `Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Text_IO` (`a-szuzti.ads`)

This package provides subprograms for Text_IO for unbounded wide wide strings, avoiding the necessity for an intermediate operation with ordinary wide wide strings.

12.19 `Ada.Task_Initialization` (`a-tasini.ads`)

This package provides a way to set a global initialization handler that is automatically invoked whenever a task is activated. Handlers are parameterless procedures. Note that such a handler is only invoked for those tasks activated after the handler is set.

12.20 `Ada.Text_IO.C_Streams` (`a-tiocst.ads`)

This package provides subprograms that allow interfacing between C streams and `Text_IO`. The stream identifier can be extracted from a file opened on the Ada side, and an Ada file can be constructed from a stream opened on the C side.

12.21 `Ada.Text_IO.Reset_Standard_Files` (`a-tirsfi.ads`)

This procedure is used to reset the status of the standard files used by Ada.Text_IO. This is useful in a situation (such as a restart in an embedded application) where the status of the files may change during execution (for example a standard input file may be redefined to be interactive).

12.22 Ada.Wide_Characters.Unicode (a-wichun.ads)

This package provides subprograms that allow categorization of Wide_Character values according to Unicode categories.

12.23 Ada.Wide_Text_IO.C_Streams (a-wtcstr.ads)

This package provides subprograms that allow interfacing between C streams and `Wide_Text_IO`. The stream identifier can be extracted from a file opened on the Ada side, and an Ada file can be constructed from a stream opened on the C side.

12.24 Ada.Wide_Text_IO.Reset_Standard_Files (a-wrstfi.ads)

This procedure is used to reset the status of the standard files used by Ada.Wide_Text_IO. This is useful in a situation (such as a restart in an embedded application) where the status of the files may change during execution (for example a standard input file may be redefined to be interactive).

12.25 Ada.Wide_Wide_Characters.Unicode (a-zchuni.ads)

This package provides subprograms that allow categorization of Wide_Wide_Character values according to Unicode categories.

12.26 Ada.Wide_Wide_Text_IO.C_Streams (a-ztcstr.ads)

This package provides subprograms that allow interfacing between C streams and `Wide_Wide_Text_IO`. The stream identifier can be extracted from a file opened on the Ada side, and an Ada file can be constructed from a stream opened on the C side.

12.27 Ada.Wide_Wide_Text_IO.Reset_Standard_Files (a-zrstfi.ads)

This procedure is used to reset the status of the standard files used by Ada.Wide_Wide_Text_IO. This is useful in a situation (such as a restart in an embedded application) where the status of the files may change during execution (for example a standard input file may be redefined to be interactive).

12.28 GNAT.Altivec (g-altive.ads)

This is the root package of the GNAT AltiVec binding. It provides definitions of constants and types common to all the versions of the binding.

12.29 GNAT.Altivec.Conversions (g-altcon.ads)

This package provides the Vector/View conversion routines.

12.30 GNAT.Altivec.Vector_Operations (g-alveop.ads)

This package exposes the Ada interface to the AltiVec operations on vector objects. A soft emulation is included by default in the GNAT library. The hard binding is provided as a separate package. This unit is common to both bindings.

12.31 GNAT.Altivec.Vector_Types (g-alvety.ads)

This package exposes the various vector types part of the Ada binding to AltiVec facilities.

12.32 GNAT.Altivec.Vector_Views (g-alvevi.ads)

This package provides public 'View' data types from/to which private vector representations can be converted via GNAT.Altivec.Conversions. This allows convenient access to individual vector elements and provides a simple way to initialize vector objects.

12.33 GNAT.Array_Split (g-arrspl.ads)

Useful array-manipulation routines: given a set of separators, split an array wherever the separators appear, and provide direct access to the resulting slices.

12.34 GNAT.AWK (g-awk.ads)

Provides AWK-like parsing functions, with an easy interface for parsing one or more files containing formatted data. The file is viewed as a database where each record is a line and a field is a data element in this line.

12.35 GNAT.Binary_Search (g-binsea.ads)

Allow binary search of a sorted array (or of an array-like container; the generic does not reference the array directly).

12.36 GNAT.Bind_Environment (g-binenv.ads)

Provides access to key=value associations captured at bind time. These associations can be specified using the `-V` binder command line switch.

12.37 GNAT.Branch_Prediction (g-brapre.ads)

Provides routines giving hints to the branch predictor of the code generator.

12.38 GNAT.Bounded_Buffers (g-boubuf.ads)

Provides a concurrent generic bounded buffer abstraction. Instances are useful directly or as parts of the implementations of other abstractions, such as mailboxes.

12.39 GNAT.Bounded_Mailboxes (g-boumai.ads)

Provides a thread-safe asynchronous intertask mailbox communication facility.

12.40 `GNAT.Bubble_Sort` (`g-bubsor.ads`)

Provides a general implementation of bubble sort usable for sorting arbitrary data items. Exchange and comparison procedures are provided by passing access-to-procedure values.

12.41 `GNAT.Bubble_Sort_A` (`g-busora.ads`)

Provides a general implementation of bubble sort usable for sorting arbitrary data items. Move and comparison procedures are provided by passing access-to-procedure values. This is an older version, retained for compatibility. Usually `GNAT.Bubble_Sort` will be preferable.

12.42 `GNAT.Bubble_Sort_G` (`g-busorg.ads`)

Similar to `Bubble_Sort_A` except that the move and sorting procedures are provided as generic parameters, this improves efficiency, especially if the procedures can be inlined, at the expense of duplicating code for multiple instantiations.

12.43 `GNAT.Byte_Order_Mark` (`g-byorma.ads`)

Provides a routine which given a string, reads the start of the string to see whether it is one of the standard byte order marks (BOM's) which signal the encoding of the string. The routine includes detection of special XML sequences for various UCS input formats.

12.44 `GNAT.Byte_Swapping` (`g-bytswa.ads`)

General routines for swapping the bytes in 2-, 4-, and 8-byte quantities. Machine-specific implementations are available in some cases.

12.45 `GNAT.C_Time` (`g-c_time.ads`)

Provides the time_t, timeval and timespec types corresponding to the C types defined by the OS, as well as various conversion functions.

12.46 `GNAT.Calendar` (`g-calend.ads`)

Extends the facilities provided by `Ada.Calendar` to include handling of days of the week, an extended `Split` and `Time_Of` capability.

12.47 `GNAT.Calendar.Time_IO` (`g-catiio.ads`)

12.48 `GNAT.CRC32` (`g-crc32.ads`)

This package implements the CRC-32 algorithm. For a full description of this algorithm see 'Computation of Cyclic Redundancy Checks via Table Look-Up', *Communications of the ACM*, Vol. 31 No. 8, pp. 1008-1013, Aug. 1988. Sarwate, D.V.

12.49 `GNAT.Case_Util` (`g-casuti.ads`)

A set of simple routines for handling upper and lower casing of strings without the overhead of the full casing tables in `Ada.Characters.Handling`.

12.50 `GNAT.CGI` (`g-cgi.ads`)

This is a package for interfacing a GNAT program with a Web server via the Common Gateway Interface (CGI). Basically this package parses the CGI parameters, which are a set of key/value pairs sent by the Web server. It builds a table whose index is the key and provides some services to deal with this table.

12.51 `GNAT.CGI.Cookie` (`g-cgicoo.ads`)

This is a package to interface a GNAT program with a Web server via the Common Gateway Interface (CGI). It exports services to deal with Web cookies (piece of information kept in the Web client software).

12.52 `GNAT.CGI.Debug` (`g-cgideb.ads`)

This is a package to help debugging CGI (Common Gateway Interface) programs written in Ada.

12.53 `GNAT.Command_Line` (`g-comlin.ads`)

Provides a high level interface to `Ada.Command_Line` facilities, including the ability to scan for named switches with optional parameters and expand file names using wildcard notations.

12.54 `GNAT.Compiler_Version` (`g-comver.ads`)

Provides a routine for obtaining the version of the compiler used to compile the program. More accurately this is the version of the binder used to bind the program (this will normally be the same as the version of the compiler if a consistent tool set is used to compile all units of a partition).

12.55 `GNAT.Ctrl_C` (`g-ctrl_c.ads`)

Provides a simple interface to handle Ctrl-C keyboard events.

12.56 `GNAT.Current_Exception` (`g-curexc.ads`)

Provides access to information on the current exception that has been raised without the need for using the Ada 95 / Ada 2005 exception choice parameter specification syntax. This is particularly useful in simulating typical facilities for obtaining information about exceptions provided by Ada 83 compilers.

12.57 `GNAT.Debug_Pools` (`g-debpoo.ads`)

Provides a debugging storage pools that helps tracking memory corruption problems. See `The GNAT Debug_Pool Facility` section in the *GNAT User's Guide*.

12.58 `GNAT.Debug_Utilities` (`g-debuti.ads`)

Provides a few useful utilities for debugging purposes, including conversion to and from string images of address values. Supports both C and Ada formats for hexadecimal literals.

12.59 GNAT.Decode_String (g-decstr.ads)

A generic package providing routines for decoding wide character and wide wide character strings encoded as sequences of 8-bit characters using a specified encoding method. Includes validation routines, and also routines for stepping to next or previous encoded character in an encoded string. Useful in conjunction with Unicode character coding. Note there is a preinstantiation for UTF-8. See next entry.

12.60 GNAT.Decode_UTF8_String (g-deutst.ads)

A preinstantiation of GNAT.Decode_Strings for UTF-8 encoding.

12.61 GNAT.Directory_Operations (g-dirope.ads)

Provides a set of routines for manipulating directories, including changing the current directory, making new directories, and scanning the files in a directory.

12.62 GNAT.Directory_Operations.Iteration (g-diopit.ads)

A child unit of GNAT.Directory_Operations providing additional operations for iterating through directories.

12.63 GNAT.Dynamic_HTables (g-dynhta.ads)

A generic implementation of hash tables that can be used to hash arbitrary data. Provided in two forms, a simple form with built in hash functions, and a more complex form in which the hash function is supplied.

This package provides a facility similar to that of `GNAT.HTable`, except that this package declares a type that can be used to define dynamic instances of the hash table, while an instantiation of `GNAT.HTable` creates a single instance of the hash table.

12.64 GNAT.Dynamic_Tables (g-dyntab.ads)

A generic package providing a single dimension array abstraction where the length of the array can be dynamically modified.

This package provides a facility similar to that of `GNAT.Table`, except that this package declares a type that can be used to define dynamic instances of the table, while an instantiation of `GNAT.Table` creates a single instance of the table type.

12.65 GNAT.Encode_String (g-encstr.ads)

A generic package providing routines for encoding wide character and wide wide character strings as sequences of 8-bit characters using a specified encoding method. Useful in conjunction with Unicode character coding. Note there is a preinstantiation for UTF-8. See next entry.

12.66 GNAT.Encode_UTF8_String (g-enutst.ads)

A preinstantiation of GNAT.Encode_Strings for UTF-8 encoding.

12.67 GNAT.Exception_Actions (g-excact.ads)

Provides callbacks when an exception is raised. Callbacks can be registered for specific exceptions, or when any exception is raised. This can be used for instance to force a core dump to ease debugging.

12.68 GNAT.Exception_Traces (g-exctra.ads)

Provides an interface allowing to control automatic output upon exception occurrences.

12.69 GNAT.Exceptions (g-except.ads)

Normally it is not possible to raise an exception with a message from a subprogram in a pure package, since the necessary types and subprograms are in `Ada.Exceptions` which is not a pure unit. `GNAT.Exceptions` provides a facility for getting around this limitation for a few predefined exceptions, and for example allows raising `Constraint_Error` with a message from a pure subprogram.

12.70 GNAT.Expect (g-expect.ads)

Provides a set of subprograms similar to what is available with the standard Tcl Expect tool. It allows you to easily spawn and communicate with an external process. You can send commands or inputs to the process, and compare the output with some expected regular expression. Currently `GNAT.Expect` is implemented on all native GNAT ports. It is not implemented for cross ports, and in particular is not implemented for VxWorks or LynxOS.

12.71 GNAT.Expect.TTY (g-exptty.ads)

As GNAT.Expect but using pseudo-terminal. Currently `GNAT.Expect.TTY` is implemented on all native GNAT ports. It is not implemented for cross ports, and in particular is not implemented for VxWorks or LynxOS.

12.72 GNAT.Float_Control (g-flocon.ads)

Provides an interface for resetting the floating-point processor into the mode required for correct semantic operation in Ada. Some third party library calls may cause this mode to be modified, and the Reset procedure in this package can be used to reestablish the required mode.

12.73 GNAT.Formatted_String (g-forstr.ads)

Provides support for C/C++ printf() formatted strings. The format is copied from the printf() routine and should therefore give identical output. Some generic routines are provided to be able to use types derived from Integer, Float or enumerations as values for the formatted string.

12.74 GNAT.Generic_Fast_Math_Functions (g-gfmafu.ads)

Provides direct access to the underlying implementation of the common mathematical functions, generally from the system mathematical library. This differs from

`Ada.Numerics.Generic_Elementary_Functions` in that the implementation may deviate from the semantics specified for these functions in the Reference Manual, for example `Numerics.Argument_Error` is not raised. On selected platforms, some of these functions may also have a vector implementation that can be automatically used by the compiler when auto-vectorization is enabled.

12.75 `GNAT.Heap_Sort` (`g-heasor.ads`)

Provides a general implementation of heap sort usable for sorting arbitrary data items. Exchange and comparison procedures are provided by passing access-to-procedure values. The algorithm used is a modified heap sort that performs approximately N*log(N) comparisons in the worst case.

12.76 `GNAT.Heap_Sort_A` (`g-hesora.ads`)

Provides a general implementation of heap sort usable for sorting arbitrary data items. Move and comparison procedures are provided by passing access-to-procedure values. The algorithm used is a modified heap sort that performs approximately N*log(N) comparisons in the worst case. This differs from `GNAT.Heap_Sort` in having a less convenient interface, but may be slightly more efficient.

12.77 `GNAT.Heap_Sort_G` (`g-hesorg.ads`)

Similar to `Heap_Sort_A` except that the move and sorting procedures are provided as generic parameters, this improves efficiency, especially if the procedures can be inlined, at the expense of duplicating code for multiple instantiations.

12.78 `GNAT.HTable` (`g-htable.ads`)

A generic implementation of hash tables that can be used to hash arbitrary data. Provides two approaches, one a simple static approach, and the other allowing arbitrary dynamic hash tables.

12.79 `GNAT.IO` (`g-io.ads`)

A simple preelaborable input-output package that provides a subset of simple Text_IO functions for reading characters and strings from Standard_Input, and writing characters, strings and integers to either Standard_Output or Standard_Error.

12.80 `GNAT.IO_Aux` (`g-io_aux.ads`)

Provides some auxiliary functions for use with Text_IO, including a test for whether a file exists, and functions for reading a line of text.

12.81 `GNAT.Lock_Files` (`g-locfil.ads`)

Provides a general interface for using files as locks. Can be used for providing program level synchronization.

12.82 GNAT.MBBS_Discrete_Random (g-mbdira.ads)

The original implementation of `Ada.Numerics.Discrete_Random`. Uses a modified version of the Blum-Blum-Shub generator.

12.83 GNAT.MBBS_Float_Random (g-mbflra.ads)

The original implementation of `Ada.Numerics.Float_Random`. Uses a modified version of the Blum-Blum-Shub generator.

12.84 GNAT.MD5 (g-md5.ads)

Implements the MD5 Message-Digest Algorithm as described in RFC 1321, and the HMAC-MD5 message authentication function as described in RFC 2104 and FIPS PUB 198.

12.85 GNAT.Memory_Dump (g-memdum.ads)

Provides a convenient routine for dumping raw memory to either the standard output or standard error files. Uses GNAT.IO for actual output.

12.86 GNAT.Most_Recent_Exception (g-moreex.ads)

Provides access to the most recently raised exception. Can be used for various logging purposes, including duplicating functionality of some Ada 83 implementation dependent extensions.

12.87 GNAT.OS_Lib (g-os_lib.ads)

Provides a range of target independent operating system interface functions, including time/date management, file operations, subprocess management, including a portable spawn procedure, and access to environment variables and error return codes.

12.88 GNAT.Perfect_Hash_Generators (g-pehage.ads)

Provides a generator of static minimal perfect hash functions. No collisions occur and each item can be retrieved from the table in one probe (perfect property). The hash table size corresponds to the exact size of the key set and no larger (minimal property). The key set has to be known in advance (static property). The hash functions are also order preserving. If w2 is inserted after w1 in the generator, their hashcode are in the same order. These hashing functions are very convenient for use with realtime applications.

12.89 GNAT.Random_Numbers (g-rannum.ads)

Provides random number capabilities which extend those available in the standard Ada library and are more convenient to use. This package is however NOT suitable for situations requiring cryptographically secure randomness.

12.90 GNAT.Regexp (g-regexp.ads)

A simple implementation of regular expressions, using a subset of regular expression syntax copied from familiar Unix style utilities. This is the simplest of the three pattern matching packages provided, and is particularly suitable for 'file globbing' applications.

12.91 GNAT.Registry (g-regist.ads)

This is a high level binding to the Windows registry. It is possible to do simple things like reading a key value, creating a new key. For full registry API, but at a lower level of abstraction, refer to the Win32.Winreg package provided with the Win32Ada binding

12.92 GNAT.Regpat (g-regpat.ads)

A complete implementation of Unix-style regular expression matching, copied from the original V7 style regular expression library written in C by Henry Spencer (and binary compatible with this C library).

12.93 GNAT.Rewrite_Data (g-rewdat.ads)

A unit to rewrite on-the-fly string occurrences in a stream of data. The implementation has a very minimal memory footprint as the full content to be processed is not loaded into memory all at once. This makes this interface usable for large files or socket streams.

12.94 GNAT.Secondary_Stack_Info (g-sestin.ads)

Provides the capability to query the high water mark of the current task's secondary stack.

12.95 GNAT.Semaphores (g-semaph.ads)

Provides classic counting and binary semaphores using protected types.

12.96 GNAT.Serial_Communications (g-sercom.ads)

Provides a simple interface to send and receive data over a serial port. This is only supported on GNU/Linux and Windows.

12.97 GNAT.SHA1 (g-sha1.ads)

Implements the SHA-1 Secure Hash Algorithm as described in FIPS PUB 180-3 and RFC 3174, and the HMAC-SHA1 message authentication function as described in RFC 2104 and FIPS PUB 198.

12.98 GNAT.SHA224 (g-sha224.ads)

Implements the SHA-224 Secure Hash Algorithm as described in FIPS PUB 180-3, and the HMAC-SHA224 message authentication function as described in RFC 2104 and FIPS PUB 198.

12.99 GNAT.SHA256 (g-sha256.ads)

Implements the SHA-256 Secure Hash Algorithm as described in FIPS PUB 180-3, and the HMAC-SHA256 message authentication function as described in RFC 2104 and FIPS PUB 198.

12.100 GNAT.SHA384 (g-sha384.ads)

Implements the SHA-384 Secure Hash Algorithm as described in FIPS PUB 180-3, and the HMAC-SHA384 message authentication function as described in RFC 2104 and FIPS PUB 198.

12.101 GNAT.SHA512 (g-sha512.ads)

Implements the SHA-512 Secure Hash Algorithm as described in FIPS PUB 180-3, and the HMAC-SHA512 message authentication function as described in RFC 2104 and FIPS PUB 198.

12.102 GNAT.Signals (g-signal.ads)

Provides the ability to manipulate the blocked status of signals on supported targets.

12.103 GNAT.Sockets (g-socket.ads)

A high level and portable interface to develop sockets based applications. This package is based on the sockets thin binding found in `GNAT.Sockets.Thin`. Currently `GNAT.Sockets` is implemented on all native GNAT ports and on VxWorks cross ports. It is not implemented for the LynxOS cross port.

12.104 GNAT.Source_Info (g-souinf.ads)

Provides subprograms that give access to source code information known at compile time, such as the current file name and line number. Also provides subprograms yielding the date and time of the current compilation (like the C macros `__DATE__` and `__TIME__`)

12.105 GNAT.Spelling_Checker (g-speche.ads)

Provides a function for determining whether one string is a plausible near misspelling of another string.

12.106 GNAT.Spelling_Checker_Generic (g-spchge.ads)

Provides a generic function that can be instantiated with a string type for determining whether one string is a plausible near misspelling of another string.

12.107 GNAT.Spitbol.Patterns (g-spipat.ads)

A complete implementation of SNOBOL4 style pattern matching. This is the most elaborate of the pattern matching packages provided. It fully duplicates the SNOBOL4 dynamic pattern construction and matching capabilities, using the efficient algorithm developed by Robert Dewar for the SPITBOL system.

12.108 GNAT.Spitbol (g-spitbo.ads)

The top level package of the collection of SPITBOL-style functionality, this package provides basic SNOBOL4 string manipulation functions, such as Pad, Reverse, Trim, Substr capability, as well as a generic table function useful for constructing arbitrary mappings from strings in the style of the SNOBOL4 TABLE function.

12.109 GNAT.Spitbol.Table_Boolean (g-sptabo.ads)

A library level of instantiation of `GNAT.Spitbol.Patterns.Table` for type `Standard.Boolean`, giving an implementation of sets of string values.

12.110 GNAT.Spitbol.Table_Integer (g-sptain.ads)

A library level of instantiation of `GNAT.Spitbol.Patterns.Table` for type `Standard.Integer`, giving an implementation of maps from string to integer values.

12.111 GNAT.Spitbol.Table_VString (g-sptavs.ads)

A library level of instantiation of `GNAT.Spitbol.Patterns.Table` for a variable length string type, giving an implementation of general maps from strings to strings.

12.112 GNAT.SSE (g-sse.ads)

Root of a set of units aimed at offering Ada bindings to a subset of the Intel(r) Streaming SIMD Extensions with GNAT on the x86 family of targets. It exposes vector component types together with a general introduction to the binding contents and use.

12.113 GNAT.SSE.Vector_Types (g-ssvety.ads)

SSE vector types for use with SSE related intrinsics.

12.114 GNAT.String_Hash (g-strhas.ads)

Provides a generic hash function working on arrays of scalars. Both the scalar type and the hash result type are parameters.

12.115 GNAT.Strings (g-string.ads)

Common String access types and related subprograms. Basically it defines a string access and an array of string access types.

12.116 GNAT.String_Split (g-strspl.ads)

Useful string manipulation routines: given a set of separators, split a string wherever the separators appear, and provide direct access to the resulting slices. This package is instantiated from `GNAT.Array_Split`.

12.117 GNAT.Table (g-table.ads)

A generic package providing a single dimension array abstraction where the length of the array can be dynamically modified.

This package provides a facility similar to that of `GNAT.Dynamic_Tables`, except that this package declares a single instance of the table type, while an instantiation of `GNAT.Dynamic_Tables` creates a type that can be used to define dynamic instances of the table.

12.118 GNAT.Task_Lock (g-tasloc.ads)

A very simple facility for locking and unlocking sections of code using a single global task lock. Appropriate for use in situations where contention between tasks is very rarely expected.

12.119 GNAT.Time_Stamp (g-timsta.ads)

Provides a simple function that returns a string YYYY-MM-DD HH:MM:SS.SS that represents the current date and time in ISO 8601 format. This is a very simple routine with minimal code and there are no dependencies on any other unit.

12.120 GNAT.Threads (g-thread.ads)

Provides facilities for dealing with foreign threads which need to be known by the GNAT run-time system. Consult the documentation of this package for further details if your program has threads that are created by a non-Ada environment which then accesses Ada code.

12.121 GNAT.Traceback (g-traceb.ads)

Provides a facility for obtaining non-symbolic traceback information, useful in various debugging situations.

12.122 GNAT.Traceback.Symbolic (g-trasym.ads)

12.123 GNAT.UTF_32 (g-utf_32.ads)

This is a package intended to be used in conjunction with the `Wide_Character` type in Ada 95 and the `Wide_Wide_Character` type in Ada 2005 (available in `GNAT` in Ada 2005 mode). This package contains Unicode categorization routines, as well as lexical categorization routines corresponding to the Ada 2005 lexical rules for identifiers and strings, and also a lower case to upper case fold routine corresponding to the Ada 2005 rules for identifier equivalence.

12.124 GNAT.UTF_32_Spelling_Checker (g-u3spch.ads)

Provides a function for determining whether one wide wide string is a plausible near misspelling of another wide wide string, where the strings are represented using the UTF_32_String type defined in System.Wch_Cnv.

12.125 GNAT.Wide_Spelling_Checker (g-wispch.ads)

Provides a function for determining whether one wide string is a plausible near misspelling of another wide string.

12.126 GNAT.Wide_String_Split (g-wistsp.ads)

Useful wide string manipulation routines: given a set of separators, split a wide string wherever the separators appear, and provide direct access to the resulting slices. This package is instantiated from `GNAT.Array_Split`.

12.127 GNAT.Wide_Wide_Spelling_Checker (g-zspche.ads)

Provides a function for determining whether one wide wide string is a plausible near misspelling of another wide wide string.

12.128 GNAT.Wide_Wide_String_Split (g-zistsp.ads)

Useful wide wide string manipulation routines: given a set of separators, split a wide wide string wherever the separators appear, and provide direct access to the resulting slices. This package is instantiated from `GNAT.Array_Split`.

12.129 Interfaces.C.Extensions (i-cexten.ads)

This package contains additional C-related definitions, intended for use with either manually or automatically generated bindings to C libraries.

12.130 Interfaces.C.Streams (i-cstrea.ads)

This package is a binding for the most commonly used operations on C streams.

12.131 Interfaces.Packed_Decimal (i-pacdec.ads)

This package provides a set of routines for conversions to and from a packed decimal format compatible with that used on IBM mainframes.

12.132 Interfaces.VxWorks (i-vxwork.ads)

This package provides a limited binding to the VxWorks API.

12.133 Interfaces.VxWorks.IO (i-vxwoio.ads)

This package provides a binding to the ioctl (IO/Control) function of VxWorks, defining a set of option values and function codes. A particular use of this package is to enable the use of Get_Immediate under VxWorks.

12.134 System.Address_Image (s-addima.ads)

This function provides a useful debugging function that gives an (implementation dependent) string which identifies an address.

12.135 System.Assertions (s-assert.ads)

This package provides the declaration of the exception raised by an run-time assertion failure, as well as the routine that is used internally to raise this assertion.

12.136 System.Atomic_Counters (s-atocou.ads)

This package provides the declaration of an atomic counter type, together with efficient routines (using hardware synchronization primitives) for incrementing, decrementing, and testing of these counters. This package is implemented on most targets, including all Alpha, AARCH64, ARM, ia64, PowerPC, SPARC V9, x86, and x86_64 platforms.

12.137 System.Memory (s-memory.ads)

This package provides the interface to the low level routines used by the generated code for allocation and freeing storage for the default storage pool (analogous to the C routines malloc and free). It also provides a reallocation interface analogous to the C routine realloc. The body of this unit may be modified to provide alternative allocation mechanisms for the default pool, and in addition, direct calls to this unit may be made for low level allocation uses (for example see the body of `GNAT.Tables`).

12.138 System.Multiprocessors (s-multip.ads)

This is an Ada 2012 unit defined in the Ada 2012 Reference Manual, but in GNAT we also make it available in Ada 95 and Ada 2005 (where it is technically an implementation-defined addition).

12.139 System.Multiprocessors.Dispatching_Domains (s-mudido.ads)

This is an Ada 2012 unit defined in the Ada 2012 Reference Manual, but in GNAT we also make it available in Ada 95 and Ada 2005 (where it is technically an implementation-defined addition).

12.140 System.Partition_Interface (s-parint.ads)

This package provides facilities for partition interfacing. It is used primarily in a distribution context when using Annex E with `PolyORB`. 'NB!' See the note in [PolyORB], page 372, regarding the lifetime of this product.

12.141 System.Pool_Global (s-pooglo.ads)

This package provides a storage pool that is equivalent to the default storage pool used for access types for which no pool is specifically declared. It uses malloc/free to allocate/free and does not attempt to do any automatic reclamation.

12.142 System.Pool_Local (s-pooloc.ads)

This package provides a storage pool that is intended for use with locally defined access types. It uses malloc/free for allocate/free, and maintains a list of allocated blocks, so that all storage allocated for the pool can be freed automatically when the pool is finalized.

12.143 System.Restrictions (s-restri.ads)

This package provides facilities for accessing at run time the status of restrictions specified at compile time for the partition. Information is available both with regard to actual restrictions specified, and with regard to compiler determined information on which restrictions are violated by one or more packages in the partition.

12.144 System.Rident (s-rident.ads)

This package provides definitions of the restrictions identifiers supported by GNAT, and also the format of the restrictions provided in package System.Restrictions. It is not normally necessary to `with` this generic package since the necessary instantiation is included in package System.Restrictions.

12.145 System.Strings.Stream_Ops (s-ststop.ads)

This package provides a set of stream subprograms for standard string types. It is intended primarily to support implicit use of such subprograms when stream attributes are applied to string types, but the subprograms in this package can be used directly by application programs.

12.146 System.Unsigned_Types (s-unstyp.ads)

This package contains definitions of standard unsigned types that correspond in size to the standard signed types declared in Standard, and (unlike the types in Interfaces) have corresponding names. It also contains some related definitions for other specialized types used by the compiler in connection with packed array types.

12.147 System.Wch_Cnv (s-wchcnv.ads)

This package provides routines for converting between wide and wide wide characters and a representation as a value of type `Standard.String`, using a specified wide character encoding method. It uses definitions in package `System.Wch_Con`.

12.148 System.Wch_Con (s-wchcon.ads)

This package provides definitions and descriptions of the various methods used for encoding wide characters in ordinary strings. These definitions are used by the package `System.Wch_Cnv`.

13 Interfacing to Other Languages

The facilities in Annex B of the Ada Reference Manual are fully implemented in GNAT, and in addition, a full interface to C++ is provided.

13.1 Interfacing to C

Interfacing to C with GNAT can use one of two approaches:

* The types in the package `Interfaces.C` may be used.
* Standard Ada types may be used directly. This may be less portable to other compilers, but will work on all GNAT compilers, which guarantee correspondence between the C and Ada types.

Pragma `Convention C` may be applied to Ada types, but mostly has no effect, since this is the default. The following table shows the correspondence between Ada scalar types and the corresponding C types.

Ada Type	**C Type**
`Integer`	`int`
`Short_Integer`	`short`
`Short_Short_Integer`	`signed char`
`Long_Integer`	`long`
`Long_Long_Integer`	`long long`
`Short_Float`	`float`
`Float`	`float`
`Long_Float`	`double`
`Long_Long_Float`	This is the longest floating-point type supported by the hardware.

Additionally, there are the following general correspondences between Ada and C types:

* Ada enumeration types map to C enumeration types directly if pragma `Convention C` is specified, which causes them to have a length of 32 bits, except for boolean types which map to C99 `bool` and for which the length is 8 bits. Without pragma `Convention C`, Ada enumeration types map to 8, 16, or 32 bits (i.e., C types `signed char`, `short`, `int`, respectively) depending on the number of values passed. This is the only case in which pragma `Convention C` affects the representation of an Ada type.
* Ada access types map to C pointers, except for the case of pointers to unconstrained types in Ada, which have no direct C equivalent.

* Ada arrays map directly to C arrays.
* Ada records map directly to C structures.
* Packed Ada records map to C structures where all members are bit fields of the length corresponding to the `type'Size` value in Ada.

13.2 Interfacing to C++

The interface to C++ makes use of the following pragmas, which are primarily intended to be constructed automatically using a binding generator tool, although it is possible to construct them by hand.

Using these pragmas it is possible to achieve complete inter-operability between Ada tagged types and C++ class definitions. See [Implementation Defined Pragmas], page 4, for more details.

`pragma CPP_Class ([Entity =>] *LOCAL_NAME*)`
: The argument denotes an entity in the current declarative region that is declared as a tagged or untagged record type. It indicates that the type corresponds to an externally declared C++ class type, and is to be laid out the same way that C++ would lay out the type.

  Note: Pragma `CPP_Class` is currently obsolete. It is supported for backward compatibility but its functionality is available using pragma `Import` with `Convention` = `CPP`.

`pragma CPP_Constructor ([Entity =>] *LOCAL_NAME*)`
: This pragma identifies an imported function (imported in the usual way with pragma `Import`) as corresponding to a C++ constructor.

A few restrictions are placed on the use of the `Access` attribute in conjunction with subprograms subject to convention `CPP`: the attribute may be used neither on primitive operations of a tagged record type with convention `CPP`, imported or not, nor on subprograms imported with pragma `CPP_Constructor`.

In addition, C++ exceptions are propagated and can be handled in an `others` choice of an exception handler. The corresponding Ada occurrence has no message, and the simple name of the exception identity contains `Foreign_Exception`. Finalization and awaiting dependent tasks works properly when such foreign exceptions are propagated.

It is also possible to import a C++ exception using the following syntax:

```
LOCAL_NAME : exception;
pragma Import (Cpp,
  [Entity =>] LOCAL_NAME,
  [External_Name =>] static_string_EXPRESSION);
```

The `External_Name` is the name of the C++ RTTI symbol. You can then cover a specific C++ exception in an exception handler. If the string ends with "'Class", as if referencing the Class attribute of the C++ type, that enables "class-wide" type matching, i.e., instances of C++ classes derived from the one denoted by the RTTI symbol, that would be caught by C++ handlers for that type, will also be caught by Ada handlers for `Entity`. For non-class-wide RTTI symbols imported from C++, only exact type matches will be handled. C++

rethrown (dependent) exceptions are not distinguishable from the corresponding primary exceptions: they are handled exactly as if the primary exception had been raised.

With imported exceptions, especially with base-type matching, a single handled_sequence_of_statements may have exception handlers with choices that cover the same C++ types in ways that GNAT cannot detect. For example, C++ classes `base` and `derived` may be imported as exceptions with base-type matching, but GNAT does not know that they are related by inheritance, only the runtime will know it. Given:

```
exception
  when Derived_Exception => null;
  when Base_Exception => null;
  when others => null;
```

the earliest handler that matches the type of the raised object will be selected. If an instance of `derived` or a further derived type is raised, the first handler will be used; if an instance of `base` that is not an instance of `derived` is raised, the second handler will be used; raised objects that are not instances of `base` will be handled by the `others` handler. However, if the handlers were reordered (`others` must remain last), the `Derived_Exception` handler would never be used, because `Base_Exception` would match any instances of `derived` before `Derived_Exception` or `others` handlers were considered. Mixing exact-type and base-type matching exceptions may also involve overlapping handlers that GNAT will not reject: an exact-type `Base_Only_Exception` handler placed before `Base_Exception` will handle instances of `base`, whereas instances of derived types will be handled by `Base_Exception`. Swapping them will cause `Base_Exception` to handle all instances of `base` and derived types, so that a subsequent handler for `Base_Only_Exception` will never be selected.

The C++ object associated with a C++ `Exception_Occurrence` may be obtained by calling the `GNAT.CPP_Exceptions.Get_Object_Address` function. There are convenience generic wrappers named `Get_Object`, `Get_Access_To_Object`, and `Get_Access_To_Tagged_Object`, parameterized on the expected Ada type. Note that, for exceptions imported from C++, the address of the object is that of the subobject of the type associated with the exception, which may have a different address from that of the full object; for C++ exceptions handled by `others` handlers, however, the address of the full object is returned.

E.g., if the imported exception uses the RTTI symbol for the base class, followed by "'Class", and the C++ code raises (throws) an instance of a derived class, a handler for that imported exception will catch this `Exception_Occurrence`, and `Get_Object_Address` will return the address of the base subobject of the raised derived object; `Get_Object`, `Get_Access_To_Object` and `Get_Access_To_Tagged_Object` only convert that address to the parameterized type, so the specified type ought to be a type that imports the C++ type whose RTTI symbol was named in the declared exception, i.e., base, not derived or any other type. GNAT cannot detect or report if a type is named that does not match the handler's RTTI-specified type.

For `others` handlers, and for exact type matches, the full object is obtained. The `Get_Type_Info` function that takes an `Exception_Occurrence` argument can be used to verify the type of the C++ object raised as an exception. The other `Get_Type_Info` function, that takes an `Exception_Id`, obtains the type expected by the handler, and no such type exists for `others` handlers. `GNAT.CPP.Std.Name` can then convert the opaque `GNAT.CPP.Std.Type_Info_Ptr` access to `std::type_info` objects, returned by either `Get_Type_Info` function, to a C++ mangled type name.

If an `Exception_Occurrence` was raised from C++, or following C++ conventions, `GNAT.Exception_Actions.Exception_Language` will return `EL_Cpp`, whether the exception handler is an imported C++ exception or `others`. `GNAT.Exception_Actions.Is_Foreign_Exception` returns True for all of these, as well as for any case in which `Exception_Language` is not `EL_Ada`.

```
--  Given the following partial package specification:

  Base_Exception : exception;
  pragma Import (Cpp, Base_Exception, "_ZTI4base'Class");
  --  Handle instances of base, and of subclasses.

  type Base is limited tagged record
    [...]
  end record;
  pragma Import (Cpp, Base);

  type Derived is limited tagged record
    [...]
  end record;
  pragma Import (Cpp, Derived);

  type Unrelated is access procedure (B : Boolean);

  function Get_Base_Obj_Acc is
    new Get_Access_To_Tagged_Object (Base);
  function Get_Derived_Obj_Acc is
    new Get_Access_To_Tagged_Object (Derived);
  function Get_Unrelated_Obj_Acc is
    new Get_Access_To_Object (Unrelated);

  procedure Raise_Derived;
  --  Raises an instance of derived (with a base subobject).


--  The comments next to each statement indicate the behavior of
--  the following pseudocode blocks:

begin
  Raise_Derived;
exception
  when BEx : Base_Exception =>
    ?? := Is_Foreign_Exception (BEx);  --  True
    ?? := Exception_Language (BEx);    --  EL_Cpp
    ?? := Name (Get_Type_Info (BEx));  --  "7derived"
    ?? := Name (Get_Type_Info (Exception_Identity (BEx)));  --  "4base"
    ?? := Get_Object_Address (BEx);    --  base subobject in derived object
```

```
      ?? := Get_Base_Obj_Acc (BEx):      --  ditto, as access to Base
      ?? := Get_Derived_Obj_Acc (BEx):   --  ditto, NO ERROR DETECTED!
      ?? := Get_Unrelated_Obj_Acc (BEx): --  ditto, NO ERROR DETECTED!
  end;


  begin
    Raise_Derived;
  exception
    when BEx : others =>
      ?? := Is_Foreign_Exception (BEx);  --  True
      ?? := Exception_Language (BEx);    --  EL_Cpp
      ?? := Name (Get_Type_Info (BEx));  --  "7derived"
      ?? := Get_Type_Info (Exception_Identity (BEx));  --  null
      ?? := Get_Object_Address (BEx);    --  full derived object
      ?? := Get_Derived_Obj_Acc (BEx):   --  ditto, as access to Derived
      ?? := Get_Base_Obj_Acc (BEx):      --  ditto, NO ERROR DETECTED!
      ?? := Get_Unrelated_Obj_Acc (BEx): --  ditto, NO ERROR DETECTED!
  end;
```

The calls marked with `NO ERROR DETECTED!` will compile sucessfully, even though the types specified in the specializations of the generic function do not match the type of the exception object that the function is expected to return. Mismatches between derived and base types are particularly relevant because they will appear to work as long as there isn't any offset between pointers to these types. This may hold in many cases, but is subject to change with various possible changes to the derived class.

The `GNAT.CPP.Std` package offers interfaces corresponding to the C++ standard type `std::type_info`. Function `To_Type_Info_Ptr` builds an opaque `Type_Info_Ptr` to reference a `std::type_info` object at a given `System.Address`.

13.3 Interfacing to COBOL

Interfacing to COBOL is achieved as described in section B.4 of the Ada Reference Manual.

13.4 Interfacing to Fortran

Interfacing to Fortran is achieved as described in section B.5 of the Ada Reference Manual. The pragma `Convention Fortran`, applied to a multi-dimensional array causes the array to be stored in column-major order as required for convenient interface to Fortran.

13.5 Interfacing to non-GNAT Ada code

It is possible to specify the convention `Ada` in a pragma `Import` or pragma `Export`. However this refers to the calling conventions used by GNAT, which may or may not be similar enough to those used by some other Ada 83 / Ada 95 / Ada 2005 compiler to allow interoperation.

If arguments types are kept simple, and if the foreign compiler generally follows system calling conventions, then it may be possible to integrate files compiled by other Ada compilers, provided that the elaboration issues are adequately addressed (for example by eliminating the need for any load time elaboration).

In particular, GNAT running on VMS is designed to be highly compatible with the DEC Ada 83 compiler, so this is one case in which it is possible to import foreign units of this type, provided that the data items passed are restricted to simple scalar values or simple record types without variants, or simple array types with fixed bounds.

14 Specialized Needs Annexes

Ada 95, Ada 2005, Ada 2012, and Ada 2022 define a number of Specialized Needs Annexes, which are not required in all implementations. However, as described in this chapter, GNAT implements all of these annexes:

'Systems Programming (Annex C)'
: The Systems Programming Annex is fully implemented.

'Real-Time Systems (Annex D)'
: The Real-Time Systems Annex is fully implemented.

'Distributed Systems (Annex E)'
: Stub generation is fully implemented in the GNAT compiler. In addition, a complete compatible PCS is available as part of `PolyORB`, a separate product. 'NB!' See the note in [PolyORB], page 372, regarding the lifetime of this product.

'Information Systems (Annex F)'
: The Information Systems annex is fully implemented.

'Numerics (Annex G)'
: The Numerics Annex is fully implemented.

'Safety and Security / High-Integrity Systems (Annex H)'
: The Safety and Security Annex (termed the High-Integrity Systems Annex in Ada 2005) is fully implemented.

15 Implementation of Specific Ada Features

This chapter describes the GNAT implementation of several Ada language facilities.

15.1 Machine Code Insertions

Package `Machine_Code` provides machine code support as described in the Ada Reference Manual in two separate forms:

* Machine code statements, consisting of qualified expressions that fit the requirements of RM section 13.8.
* An intrinsic callable procedure, providing an alternative mechanism of including machine instructions in a subprogram.

The two features are similar, and both are closely related to the mechanism provided by the asm instruction in the GNU C compiler. Full understanding and use of the facilities in this package requires understanding the asm instruction, see the section on Extended Asm in *Using_the_GNU_Compiler_Collection_(GCC)*.

Calls to the function `Asm` and the procedure `Asm` have identical semantic restrictions and effects as described below. Both are provided so that the procedure call can be used as a statement, and the function call can be used to form a code_statement.

Consider this C `asm` instruction:

```
asm ("fsinx %1 %0" : "=f" (result) : "f" (angle));
```

The equivalent can be written for GNAT as:

```
Asm ("fsinx %1 %0",
     My_Float'Asm_Output ("=f", result),
     My_Float'Asm_Input  ("f",  angle));
```

The first argument to `Asm` is the assembler template, and is identical to what is used in GNU C. This string must be a static expression. The second argument is the output operand list. It is either a single `Asm_Output` attribute reference, or a list of such references enclosed in parentheses (technically an array aggregate of such references).

The `Asm_Output` attribute denotes a function that takes two parameters. The first is a string, the second is the name of a variable of the type designated by the attribute prefix. The first (string) argument is required to be a static expression and designates the constraint (see the section on Constraints in *Using_the_GNU_Compiler_Collection_(GCC)*) for the parameter; e.g., what kind of register is required. The second argument is the variable to be written or updated with the result. The possible values for constraint are the same as those used in the RTL, and are dependent on the configuration file used to build the GCC back end. If there are no output operands, then this argument may either be omitted, or explicitly given as `No_Output_Operands`. No support is provided for GNU C's symbolic names for output parameters.

The second argument of `my_float'Asm_Output` functions as though it were an `out` parameter, which is a little curious, but all names have the form of expressions, so there is no syntactic irregularity, even though normally functions would not be permitted `out` parameters. The third argument is the list of input operands. It is either a single `Asm_Input` attribute reference, or a list of such references enclosed in parentheses (technically an array aggregate of such references).

The `Asm_Input` attribute denotes a function that takes two parameters. The first is a string, the second is an expression of the type designated by the prefix. The first (string) argument is required to be a static expression, and is the constraint for the parameter, (e.g., what kind of register is required). The second argument is the value to be used as the input argument. The possible values for the constraint are the same as those used in the RTL, and are dependent on the configuration file used to built the GCC back end. No support is provided for GNU C's symbolic names for input parameters.

If there are no input operands, this argument may either be omitted, or explicitly given as `No_Input_Operands`. The fourth argument, not present in the above example, is a list of register names, called the 'clobber' argument. This argument, if given, must be a static string expression, and is a space or comma separated list of names of registers that must be considered destroyed as a result of the `Asm` call. If this argument is the null string (the default value), then the code generator assumes that no additional registers are destroyed. In addition to registers, the special clobbers `memory` and `cc` as described in the GNU C docs are both supported.

The fifth argument, not present in the above example, called the 'volatile' argument, is by default `False`. It can be set to the literal value `True` to indicate to the code generator that all optimizations with respect to the instruction specified should be suppressed, and in particular an instruction that has outputs will still be generated, even if none of the outputs are used. See *Using_the_GNU_Compiler_Collection_(GCC)* for the full description. Generally it is strongly advisable to use Volatile for any ASM statement that is missing either input or output operands or to avoid unwanted optimizations. A warning is generated if this advice is not followed.

No support is provided for GNU C's `asm goto` feature.

The `Asm` subprograms may be used in two ways. First the procedure forms can be used anywhere a procedure call would be valid, and correspond to what the RM calls 'intrinsic' routines. Such calls can be used to intersperse machine instructions with other Ada statements. Second, the function forms, which return a dummy value of the limited private type `Asm_Insn`, can be used in code statements, and indeed this is the only context where such calls are allowed. Code statements appear as aggregates of the form:

```
Asm_Insn'(Asm (...));
Asm_Insn'(Asm_Volatile (...));
```

In accordance with RM rules, such code statements are allowed only within subprograms whose entire body consists of such statements. It is not permissible to intermix such statements with other Ada statements.

Typically the form using intrinsic procedure calls is more convenient and more flexible. The code statement form is provided to meet the RM suggestion that such a facility should be made available. The following is the exact syntax of the call to `Asm`. As usual, if named notation is used, the arguments may be given in arbitrary order, following the normal rules for use of positional and named arguments:

```
ASM_CALL ::= Asm (
                 [Template =>] static_string_EXPRESSION
               [,[Outputs  =>] OUTPUT_OPERAND_LIST      ]
               [,[Inputs   =>] INPUT_OPERAND_LIST       ]
               [,[Clobber  =>] static_string_EXPRESSION ]
```

```
                [,[Volatile =>] static_boolean_EXPRESSION] )

OUTPUT_OPERAND_LIST ::=
  [PREFIX.]No_Output_Operands
| OUTPUT_OPERAND_ATTRIBUTE
| (OUTPUT_OPERAND_ATTRIBUTE {,OUTPUT_OPERAND_ATTRIBUTE})

OUTPUT_OPERAND_ATTRIBUTE ::=
  SUBTYPE_MARK'Asm_Output (static_string_EXPRESSION, NAME)

INPUT_OPERAND_LIST ::=
  [PREFIX.]No_Input_Operands
| INPUT_OPERAND_ATTRIBUTE
| (INPUT_OPERAND_ATTRIBUTE {,INPUT_OPERAND_ATTRIBUTE})

INPUT_OPERAND_ATTRIBUTE ::=
  SUBTYPE_MARK'Asm_Input (static_string_EXPRESSION, EXPRESSION)
```

The identifiers `No_Input_Operands` and `No_Output_Operands` are declared in the package `Machine_Code` and must be referenced according to normal visibility rules. In particular if there is no `use` clause for this package, then appropriate package name qualification is required.

15.2 GNAT Implementation of Tasking

This chapter outlines the basic GNAT approach to tasking (in particular, a multi-layered library for portability) and discusses issues related to compliance with the Real-Time Systems Annex.

15.2.1 Mapping Ada Tasks onto the Underlying Kernel Threads

GNAT's run-time support comprises two layers:

* GNARL (GNAT Run-time Layer)
* GNULL (GNAT Low-level Library)

In GNAT, Ada's tasking services rely on a platform and OS independent layer known as GNARL. This code is responsible for implementing the correct semantics of Ada's task creation, rendezvous, protected operations etc.

GNARL decomposes Ada's tasking semantics into simpler lower level operations such as create a thread, set the priority of a thread, yield, create a lock, lock/unlock, etc. The spec for these low-level operations constitutes GNULLI, the GNULL Interface. This interface is directly inspired from the POSIX real-time API.

If the underlying executive or OS implements the POSIX standard faithfully, the GNULL Interface maps as is to the services offered by the underlying kernel. Otherwise, some target dependent glue code maps the services offered by the underlying kernel to the semantics expected by GNARL.

Whatever the underlying OS (VxWorks, UNIX, Windows, etc.) the key point is that each Ada task is mapped on a thread in the underlying kernel. For example, in the case of VxWorks, one Ada task = one VxWorks task.

In addition Ada task priorities map onto the underlying thread priorities. Mapping Ada tasks onto the underlying kernel threads has several advantages:

* The underlying scheduler is used to schedule the Ada tasks. This makes Ada tasks as efficient as kernel threads from a scheduling standpoint.
* Interaction with code written in C containing threads is eased since at the lowest level Ada tasks and C threads map onto the same underlying kernel concept.
* When an Ada task is blocked during I/O the remaining Ada tasks are able to proceed.
* On multiprocessor systems Ada tasks can execute in parallel.

Some threads libraries offer a mechanism to fork a new process, with the child process duplicating the threads from the parent. GNAT does not support this functionality when the parent contains more than one task.

15.2.2 Ensuring Compliance with the Real-Time Annex

Although mapping Ada tasks onto the underlying threads has significant advantages, it does create some complications when it comes to respecting the scheduling semantics specified in the real-time annex (Annex D).

For instance the Annex D requirement for the `FIFO_Within_Priorities` scheduling policy states:

> 'When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.'

While most kernels do put tasks at the end of the priority queue when a task changes its priority, (which respects the main FIFO_Within_Priorities requirement), almost none keep a thread at the beginning of its priority queue when its priority drops from the loss of inherited priority.

As a result most vendors have provided incomplete Annex D implementations.

The GNAT run-time, has a nice cooperative solution to this problem which ensures that accurate FIFO_Within_Priorities semantics are respected.

The principle is as follows. When an Ada task T is about to start running, it checks whether some other Ada task R with the same priority as T has been suspended due to the loss of priority inheritance. If this is the case, T yields and is placed at the end of its priority queue. When R arrives at the front of the queue it executes.

Note that this simple scheme preserves the relative order of the tasks that were ready to execute in the priority queue where R has been placed at the end.

15.2.3 Support for Locking Policies

This section specifies which policies specified by pragma Locking_Policy are supported on which platforms.

GNAT supports the standard `Ceiling_Locking` policy, and the implementation defined `Inheritance_Locking` and `Concurrent_Readers_Locking` policies.

`Ceiling_Locking` is supported on all platforms if the operating system supports it. In particular, `Ceiling_Locking` is not supported on VxWorks. `Inheritance_Locking` is supported on Linux, Darwin (Mac OS X), LynxOS 178, and VxWorks. `Concurrent_Readers_Locking` is supported on Linux.

Notes about `Ceiling_Locking` on Linux: If the process is running as 'root', ceiling locking is used. If the capabilities facility is installed ("sudo apt-get –assume-yes install libcap-dev" on Ubuntu, for example), and the program is linked against that library ("-largs -lcap"), and the executable file has the cap_sys_nice capability ("sudo /sbin/setcap cap_sys_nice=ep executable_file_name"), then ceiling locking is used. Otherwise, the `Ceiling_Locking` policy is ignored.

15.3 GNAT Implementation of Shared Passive Packages

GNAT fully implements the pragma `Shared_Passive` for the purpose of designating shared passive packages. This allows the use of passive partitions in the context described in the Ada Reference Manual; i.e., for communication between separate partitions of a distributed application using the features in Annex E.

However, the implementation approach used by GNAT provides for more extensive usage as follows:

'Communication between separate programs'
: This allows separate programs to access the data in passive partitions, using protected objects for synchronization where needed. The only requirement is that the two programs have a common shared file system. It is even possible for programs running on different machines with different architectures (e.g., different endianness) to communicate via the data in a passive partition.

'Persistence between program runs'
: The data in a passive package can persist from one run of a program to another, so that a later program sees the final values stored by a previous run of the same program.

The implementation approach used is to store the data in files. A separate stream file is created for each object in the package, and an access to an object causes the corresponding file to be read or written.

The environment variable `SHARED_MEMORY_DIRECTORY` should be set to the directory to be used for these files. The files in this directory have names that correspond to their fully qualified names. For example, if we have the package

```
package X is
  pragma Shared_Passive (X);
  Y : Integer;
  Z : Float;
end X;
```

and the environment variable is set to `/stemp/`, then the files created will have the names:

```
/stemp/x.y
/stemp/x.z
```

These files are created when a value is initially written to the object, and the files are retained until manually deleted. This provides the persistence semantics. If no file exists, it means that no partition has assigned a value to the variable; in this case the initial value declared in the package will be used. This model ensures that there are no issues in synchronizing the elaboration process, since elaboration of passive packages elaborates the initial values, but does not create the files.

The files are written using normal `Stream_IO` access. If you want to be able to communicate between programs or partitions running on different architectures, then you should use the XDR versions of the stream attribute routines, since these are architecture independent.

If active synchronization is required for access to the variables in the shared passive package, then as described in the Ada Reference Manual, the package may contain protected objects used for this purpose. In this case a lock file (whose name is `___lock`, with three underscores) is created in the shared memory directory.

This is used to provide the required locking semantics for proper protected object synchronization.

15.4 Code Generation for Array Aggregates

Aggregates have a rich syntax and allow the user to specify the values of complex data structures by means of a single construct. As a result, the code generated for aggregates can be quite complex and involve loops, case statements and multiple assignments. In the simplest cases, however, the compiler will recognize aggregates whose components and constraints are fully static, and in those cases the compiler will generate little or no executable code. The following is an outline of the code that GNAT generates for various aggregate constructs. For further details, you will find it useful to examine the output produced by the -gnatG flag to see the expanded source that is input to the code generator. You may also want to examine the assembly code generated at various levels of optimization.

The code generated for aggregates depends on the context, the component values, and the type. In the context of an object declaration the code generated is generally simpler than in the case of an assignment. As a general rule, static component values and static subtypes also lead to simpler code.

15.4.1 Static constant aggregates with static bounds

For the declarations:

```
type One_Dim is array (1..10) of integer;
ar0 : constant One_Dim := (1, 2, 3, 4, 5, 6, 7, 8, 9, 0);
```

GNAT generates no executable code: the constant ar0 is placed in static memory. The same is true for constant aggregates with named associations:

```
Cr1 : constant One_Dim := (4 => 16, 2 => 4, 3 => 9, 1 => 1, 5 .. 10 => 0);
Cr3 : constant One_Dim := (others => 7777);
```

The same is true for multidimensional constant arrays such as:

```
type two_dim is array (1..3, 1..3) of integer;
Unit : constant two_dim := ( (1,0,0), (0,1,0), (0,0,1));
```

The same is true for arrays of one-dimensional arrays: the following are static:

```
type ar1b  is array (1..3) of boolean;
type ar_ar is array (1..3) of ar1b;
None  : constant ar1b := (others => false);     --  fully static
```

```
None2 : constant ar_ar := (1..3 => None);       --  fully static
```

However, for multidimensional aggregates with named associations, GNAT will generate assignments and loops, even if all associations are static. The following two declarations generate a loop for the first dimension, and individual component assignments for the second dimension:

```
Zero1: constant two_dim := (1..3 => (1..3 => 0));
Zero2: constant two_dim := (others => (others => 0));
```

15.4.2 Constant aggregates with unconstrained nominal types

In such cases the aggregate itself establishes the subtype, so that associations with `others` cannot be used. GNAT determines the bounds for the actual subtype of the aggregate, and allocates the aggregate statically as well. No code is generated for the following:

```
type One_Unc is array (natural range <>) of integer;
Cr_Unc : constant One_Unc := (12,24,36);
```

15.4.3 Aggregates with static bounds

In all previous examples the aggregate was the initial (and immutable) value of a constant. If the aggregate initializes a variable, then code is generated for it as a combination of individual assignments and loops over the target object. The declarations

```
Cr_Var1 : One_Dim := (2, 5, 7, 11, 0, 0, 0, 0, 0, 0);
Cr_Var2 : One_Dim := (others > -1);
```

generate the equivalent of

```
Cr_Var1 (1) := 2;
Cr_Var1 (2) := 3;
Cr_Var1 (3) := 5;
Cr_Var1 (4) := 11;

for I in Cr_Var2'range loop
   Cr_Var2 (I) := -1;
end loop;
```

15.4.4 Aggregates with nonstatic bounds

If the bounds of the aggregate are not statically compatible with the bounds of the nominal subtype of the target, then constraint checks have to be generated on the bounds. For a multidimensional array, constraint checks may have to be applied to sub-arrays individually, if they do not have statically compatible subtypes.

15.4.5 Aggregates in assignment statements

In general, aggregate assignment requires the construction of a temporary, and a copy from the temporary to the target of the assignment. This is because it is not always possible to convert the assignment into a series of individual component assignments. For example, consider the simple case:

```
A := (A(2), A(1));
```

This cannot be converted into:

```
A(1) := A(2);
```

```
A(2) := A(1);
```

So the aggregate has to be built first in a separate location, and then copied into the target. GNAT recognizes simple cases where this intermediate step is not required, and the assignments can be performed in place, directly into the target. The following sufficient criteria are applied:

* The bounds of the aggregate are static, and the associations are static.
* The components of the aggregate are static constants, names of simple variables that are not renamings, or expressions not involving indexed components whose operands obey these rules.

If any of these conditions are violated, the aggregate will be built in a temporary (created either by the front-end or the code generator) and then that temporary will be copied onto the target.

15.5 The Size of Discriminated Records with Default Discriminants

If a discriminated type `T` has discriminants with default values, it is possible to declare an object of this type without providing an explicit constraint:

```
type Size is range 1..100;

type Rec (D : Size := 15) is record
   Name : String (1..D);
end T;

Word : Rec;
```

Such an object is said to be 'unconstrained'. The discriminant of the object can be modified by a full assignment to the object, as long as it preserves the relation between the value of the discriminant, and the value of the components that depend on it:

```
Word := (3, "yes");

Word := (5, "maybe");

Word := (5, "no"); -- raises Constraint_Error
```

In order to support this behavior efficiently, an unconstrained object is given the maximum size that any value of the type requires. In the case above, `Word` has storage for the discriminant and for a `String` of length 100. It is important to note that unconstrained objects do not require dynamic allocation. It would be an improper implementation to place on the heap those components whose size depends on discriminants. (This improper implementation was used by some Ada83 compilers, where the `Name` component above would have been stored as a pointer to a dynamic string). Following the principle that dynamic storage management should never be introduced implicitly, an Ada compiler should reserve the full size for an unconstrained declared object, and place it on the stack.

This maximum size approach has been a source of surprise to some users, who expect the default values of the discriminants to determine the size reserved for an unconstrained object: "If the default is 15, why should the object occupy a larger size?" The answer, of course, is that the discriminant may be later modified, and its full range of values must be taken into account. This is why the declaration:

```
type Rec (D : Positive := 15) is record
   Name : String (1..D);
end record;

Too_Large : Rec;
```

is flagged by the compiler with a warning: an attempt to create `Too_Large` will raise `Storage_Error`, because the required size includes `Positive'Last` bytes. As the first example indicates, the proper approach is to declare an index type of 'reasonable' range so that unconstrained objects are not too large.

One final wrinkle: if the object is declared to be `aliased`, or if it is created in the heap by means of an allocator, then it is 'not' unconstrained: it is constrained by the default values of the discriminants, and those values cannot be modified by full assignment. This is because in the presence of aliasing all views of the object (which may be manipulated by different tasks, say) must be consistent, so it is imperative that the object, once created, remain invariant.

15.6 Image Values For Nonscalar Types

Ada 2022 defines the Image, Wide_Image, and Wide_Wide image attributes for nonscalar types; earlier Ada versions defined these attributes only for scalar types. Ada RM 4.10 provides some general guidance regarding the default implementation of these attributes and the GNAT compiler follows that guidance. However, beyond that the precise details of the image text generated in these cases are deliberately not documented and are subject to change. In particular, users should not rely on formatting details (such as spaces or line breaking), record field order, image values for access types, image values for types that have ancestor or subcomponent types declared in non-Ada2022 code, image values for predefined types, or the compiler's choices regarding the implementation permissions described in Ada RM 4.10. This list is not intended to be exhaustive. If more precise control of image text is required for some type T, then T'Put_Image should be explicitly specified.

15.7 Strict Conformance to the Ada Reference Manual

The dynamic semantics defined by the Ada Reference Manual impose a set of run-time checks to be generated. By default, the GNAT compiler will insert many run-time checks into the compiled code, including most of those required by the Ada Reference Manual. However, there are two checks that are not enabled in the default mode for efficiency reasons: checks for access before elaboration on subprogram calls, and stack overflow checking (most operating systems do not perform this check by default).

Strict conformance to the Ada Reference Manual can be achieved by adding two compiler options for dynamic checks for access-before-elaboration on subprogram calls and generic instantiations ('-gnatE'), and stack overflow checking ('-fstack-check').

Note that the result of a floating point arithmetic operation in overflow and invalid situations, when the `Machine_Overflows` attribute of the result type is `False`, is to generate IEEE NaN and infinite values. This is the case for machines compliant with the IEEE floating-point standard, but on machines that are not fully compliant with this standard, such as Alpha, the '-mieee' compiler flag must be used for achieving IEEE confirming behavior (although at the cost of a significant performance penalty), so infinite and NaN values are properly generated.

16 Implementation of Ada 2022 Features

This chapter contains a complete list of Ada 2022 features that have been implemented. Generally, these features are only available if the '-gnat22' (Ada 2022 features enabled) option is set, or if the configuration pragma `Ada_2022` is used.

However, new pragmas, attributes, and restrictions are unconditionally available, since the Ada standard allows the addition of new pragmas, attributes, and restrictions (there are exceptions, which are documented in the individual descriptions), and also certain packages were made available in earlier versions of Ada.

An ISO date (YYYY-MM-DD) appears in parentheses on the description line. This date shows the implementation date of the feature. Any wavefront subsequent to this date will contain the indicated feature, as will any subsequent releases. A date of 0000-00-00 means that GNAT has always implemented the feature, or implemented it as soon as it appeared as a binding interpretation.

Each feature corresponds to an Ada Issue ('AI') approved by the Ada standardization group (ISO/IEC JTC1/SC22/WG9) for inclusion in Ada 2022.

The section "RM references" lists all modified paragraphs in the Ada 2012 reference manual. The details of each modification as well as a complete description of the AIs may be found in '`http://www.ada-auth.org/AI12-SUMMARY.HTML`'.

* 'AI12-0001 Independence and Representation clauses for atomic objects (2019-11-27)'

  The compiler accepts packing clauses in all cases, even if they have effectively no influence on the layout. Types, where packing is essentially infeasible are, for instance atomic, aliased and by-reference types.

  RM references: 13.02 (6.1/2) 13.02 (7) 13.02 (8) 13.02 (9/3) C.06 (8.1/3) C.06 (10) C.06 (11) C.06 (21) C.06 (24)

* 'AI12-0003 Specifying the standard storage pool (2020-06-25)'

  Allows the standard storage pool being specified with a `Default_Storage_Pool` pragma or aspect.

  RM references: 8.02 (11) 13.11.03 (1/3) 13.11.03 (3.1/3) 13.11.03 (4/3) 13.11.03 (4.1/3) 13.11.03 (5/3) 13.11.03 (6.2/3) 13.11.03 (6.3/3)

* 'AI12-0004 Normalization and allowed characters for identifiers (2020-06-11)'

  This AI clarifies that Ada identifiers containing characters which are not allowed in Normalization Form KC are illegal.

  RM references: 2.01 (4.1/3) 2.03 (4/3) A.03.02 (4/3) A.03.02 (32.5/3) A.03.05 (18/3) A.03.05 (51/3)

* 'AI12-0020 'Image for all types (2020-03-30)'

  Put_Image prints out a human-readable representation of an object. The functionality in Ada2022 RM is fully implemented except the support for types in the `Remote_Types` packages.

  RM references: 4.10 (0) 3.05 (27.1/2) 3.05 (27.2/2) 3.05 (27.3/2) 3.05 (27.4/2) 3.05 (27.5/2) 3.05 (27.6/2) 3.05 (27.7/2) 3.05 (28) 3.05 (29) 3.05 (30/3) 3.05 (31) 3.05 (32) 3.05 (33/3) 3.05 (37.1/2) 3.05 (38) 3.05 (39) 3.05 (43/3) 3.05 (55/3) 3.05 (55.1/5) 3.05 (55.2/4) 3.05 (55.3/4) 3.05 (55.4/4) 3.05 (59) H.04 (23) H.04 (23.8/2)

* 'AI12-0022 Raise_Expressions (2013-01-27)'

  This feature allows you to write "raise NAME [with STRING]" in an expression to rise given exception. It is particularly useful in the case of assertions such as preconditions allowing to specify which exception a precondition raises if it fails.

  RM references: 4.04 (3/3) 11.02 (6) 11.03 (2/2) 11.03 (3) 11.03 (3.1/2) 11.03 (4/2) 11.04.01 (10.1/3)

* 'AI12-0027 Access values should never designate unaliased components (2020-06-15)'

  AI12-0027 adds a requirement for a value conversion that converts from an array of unaliased components to an array of aliased components to make a copy. It defines such conversions to have a local accessibility, effectively preventing the possibility of unsafe accesses to unaliased components.

  RM references: 4.06 (24.17/3) 4.06 (24.21/2) 4.06 (58) 6.02 (10/3) 3.10.02 (10/3)

* 'AI12-0028 Import of variadic C functions (2020-03-03)'

  Ada programs can now properly call variadic C functions by means of the conventions C_Variadic_<n>, for small integer values <n>.

  RM references: B.03 (1/3) B.03 (60.15/3) B.03 (75)

* 'AI12-0030 Formal derived types and stream attribute availability (2020-08-21)'

  Corner cases involving streaming operations for formal derived limited types that are now defined to raise Program_Error. Before, behavior in these cases was undefined. Stream attribute availability is more precisely computed in cases where a derived type declaration occurs ahead of a streaming attribute specification for the parent type.

  RM references: 12.05.01 (21/3) 13.13.02 (49/2)

* 'AI12-0031 All_Calls_Remote and indirect calls (0000-00-00)'

  Remote indirect calls (i.e., calls through a remote access-to-subprogram type) behave the same as remote direct calls.

  RM references: E.02.03 (19/3)

* 'AI12-0032 Questions on 'Old (2020-04-24)'

  AI12-0032 resolves several issues related to the 'Old attribute. The GNAT compiler already implemented what the AI requires in most of those cases, but two having to do with static and dynamic checking of the accessibility level of the constant object implicitly declared for an 'Old attribute reference were not yet implemented. Accessibility checking for these constants is now implemented as defined in the AI.

  RM references: 4.01.03 (9/3) 6.01.01 (22/3) 6.01.01 (26/3) 6.01.01 (35/3)

* 'AI12-0033 Sets of CPUs when defining dispatching domains (0000-00-00)'

  The set of CPUs associated with a dispatching domain is no longer required to be a contiguous range of CPU values.

  RM references: D.16.01 (7/3) D.16.01 (9/3) D.16.01 (20/3) D.16.01 (23/3) D.16.01 (24/3) D.16.01 (26/3)

* 'AI12-0035 Accessibility checks for indefinite elements of containers (0000-00-00)'

  If the element type for an instance of one of the indefinite container generics has an access discriminant, then accessibility checks (at run-time) prevent inserting a value into a container object if the value's discriminant designates an object that is too short-lived (that is, if the designated object has an accessibility level that is deeper than that of the instance). Without this check, dangling references would be possible. GNAT handled this correctly already before this AI was issued.

RM references: A.18 (5/3) A.18.11 (8/2) A.18.12 (7/2) A.18.13 (8/2) A.18.14 (8/2) A.18.15 (4/2) A.18.16 (4/2) A.18.17 (7/3) A.18.18 (39/3) A.18.18 (47/3)

* 'AI12-0036 The actual for an untagged formal derived type cannot be tagged (2019-10-21)'

  AI12-0036 is a binding interpretation that adds the following legality rule: The actual type for a formal derived type shall be tagged if and only if the formal derived type is a private extension. The check is implemented for all Ada dialects, not just Ada 2022.

  RM references: 12.05.01 (5.1/3)

* 'AI12-0037 New types in Ada.Locales can't be converted to/from strings (2016-09-10)'

  The type definitions for Language_Code and Country_Code are now using dynamic predicates.

  RM references: A.19 (4/3)

* 'AI12-0039 Ambiguity in syntax for membership expression removed (0000-00-00)'

  An ambiguity in the syntax for membership expressions was resolved. For example, "A in B and C" can be parsed in only one way because of this AI.

  RM references: 4.04 (3/3) 4.04 (3.2/3) 4.05.02 (3.1/3) 4.05.02 (4) 4.05.02 (4.1/3) 4.05.02 (27/3) 4.05.02 (27.1/3) 4.05.02 (28.1/3) 4.05.02 (28.2/3) 4.05.02 (29/3) 4.05.02 (30/3) 4.05.02 (30.1/3) 4.05.02 (30.2/3) 4.05.02 (30.3/3) 4.09 (11/3) 4.09 (32.6/3) 8.06 (27.1/3) 3.02.04 (17/3)

* 'AI12-0040 Resolving the selecting_expression of a case_expression (0000-00-00)'

  The definition of "complete context" is corrected so that selectors of case expressions and of case statements are treated uniformly.

  RM references: 8.06 (9)

* 'AI12-0041 Type_Invariant'Class for interface types (2016-12-12)'

  Subprogram calls within class-wide type invariant expressions get resolved as primitive operations instead of being dynamically dispatched.

  RM references: 7.03.02 (1/3) 7.03.02 (3/3)

* 'AI12-0042 Type invariant checking rules (2020-06-05)'

  AI12-0042 adds rules for type invariants. Specifically, when inheriting a private dispatching operation when the ancestor operation is visible at the point of the type extension, the operation must be abstract or else overridden. In addition, for a class-wide view conversion from an object of a specific type T to which a type invariant applies, an invariant check is performed when the conversion is within the immediate scope of T.

  RM references: 7.03.02 (6/3) 7.03.02 (17/3) 7.03.02 (18/3) 7.03.02 (19/3) 7.03.02 (20/3)

* 'AI12-0043 Details of the storage pool used when Storage_Size is specified (0000-00-00)'

  Clarify that a Storage_Size specification for an access type specifies both an upper bound and a lower bound (not just a lower bound) of the amount of storage allowed for allocated objects.

  RM references: 13.11 (18)

* ‘AI12-0044 Calling visible functions from type invariant expressions (2020-05-11)’

  AI05-0289-1 extends invariant checking to *in* parameters. However, this makes it impossible to call a public function of the type from an invariant expression, as that public function will attempt to check the invariant, resulting in an infinite recursion.

  This AI specifies, that type-invariant checking is performed on parameters of mode *in* upon return from procedure calls, but not of *in*-mode parameters in functions.

  RM references: 7.03.02 (19/3)

* ‘AI12-0045 Pre- and Postconditions are allowed on generic subprograms (2015-03-17)’

  The SPARK toolset now supports contracts on generic subprograms, packages and their respective bodies.

  RM references: 6.01.01 (1/3)

* ‘AI12-0046 Enforcing legality for anonymous access components in record aggregates (0000-00-00)’

  For a record aggregate of the form (X `|` Y => ....), any relevant legality rules are checked for both for X and Y.

  For example,

  ```
  X : aliased constant String := ... ;
  type R is record
    F1 : access constant String;
    F2 : access String;
  end record;
  Obj : R := (F1 | F2 => X'Access); -- ok for F1, but illegal for F2
  ```

  RM references: 4.03.01 (16/3)

* ‘AI12-0047 Generalized iterators and discriminant-dependent components (0000-00-00)’

  Iterating over the elements of an array is subject to the same legality checks as renaming the array. For example, if an assignment to an enclosing discriminated object could cause an array object to cease to exist then we don’t allow renaming the array. So it is similarly not allowed to iterate over the elements of such an array.

  RM references: 5.05.02 (6/3)

* ‘AI12-0048 Default behavior of tasks on a multiprocessor with a specified dispatching policy (0000-00-00)’

  Clarify that if the user does not impose requirements about what CPUs a given task might execute on, then the implementation does not get to impose such requirements. This avoids potential problems with priority inversion.

  RM references: D.16.01 (30/3)

* ‘AI12-0049 Invariants need to be checked on the initialization of deferred constants (0000-00-00)’

  Invariant checking for deferred constants (and subcomponents thereof) is performed. Corrects a clear oversight in the previous RM wording.

  RM references: 7.03.02 (10/3)

* 'AI12-0050 Conformance of quantified expressions (2016-07-22)'

  Compiler rejects a subprogram body when an expression for a boolean formal parameter includes a quantified expression, and the subprogram declaration contains a textual copy of the same.

  RM references: 6.03.01 (20) 6.03.01 (21)

* 'AI12-0051 The Priority aspect can be specified when Attach_Handler is specified (0000-00-00)'

  Previous RM wording had two contradictory rules for determining (in some cases) the priority of a protected subprogram that is attached to an interrupt. The AI clarifies which one of the rules takes precedence.

  RM references: D.03 (10/3)

* 'AI12-0052 Implicit objects are considered overlapping (0000-00-00)'

  Clarify that the rules about unsynchronized concurrent access apply as one would expect in the case of predefined routines that access Text_IO's default input and default output files. There was no compiler changes needed to implement this.

  RM references: A (3/2) A.10.03 (21)

* 'AI12-0054-2 Aspect Predicate_Failure (0000-00-00)'

  New aspect Predicate_Failure is defined. A solution for the problem that a predicate like

  ```
  subtype Open_File is File with Dynamic_Predicate =\> Is_Open (Open_File) or else (raise File_Not_Open);
  ```

  does the wrong thing in the case of a membership test.

  RM references: 3.02.04 (14/3) 3.02.04 (31/3) 3.02.04 (35/3)

* 'AI12-0055 All properties of a usage profile are defined by pragmas (2020-06-09)'

  AI12-0055 allows the use of the No_Dynamic_CPU_Assignment restriction in pragmas Restrictions and Restrictions_Warnings.

  RM references: D.07 (10/3) D.13 (6/3) D.13 (8/3) D.13 (10/3)

* 'AI12-0059 Object_Size attribute (2019-12-02)'

  AI12-0059 brings GNAT-defined attribute Object_Size to Ada standard and clarifies its semantics. Given that the attribute already existed in GNAT compiler, the feature is supported for all language versions.

  RM references: 4.09.01 (2/3) 13.01 (14) 13.01 (23) 13.03 (9/3) 13.03 (50/2) 13.03 (51) 13.03 (52) 13.03 (58)

* 'AI12-0061 Iterated component associations in array aggregates (2016-09-01)'

  Ada issue AI12-061 introduces a new construct in array aggregates allowing component associations to be parameterized by a loop variable, for example:

  ```
  Array (1 .. 10) of Integer :=
    (for I in 1 .. 10 => I ** 2);
  type Matrix is
  array
    (Positive range <>, Positive range <>) of Float;
  G : constant Matrix
  :=
  ```

```
          (for I in 1 .. 4 =>
              (for J in 1 .. 4 =>
                  (if I=J then
        1.0 else 0.0))); -- Identity matrix
```

The expression in such an association can also be a function that returns a limited type, and the range can be specified by the ‘others’ choice.

RM references: 4.03.03 (5/2) 4.03.03 (6) 4.03.03 (17/3) 4.03.03 (20) 4.03.03 (23.1/4) 4.03.03 (32/3) 4.03.03 (43) 3.01 (6/3) 3.03 (6) 3.03 (18.1/3) 3.03.01 (23/3) 5.05 (6) 8.01 (2.1/4)

* ‘AI12-0062 Raise exception with failing string function (0000-00-00)’

  Clarify that if raising exception E1 is accompanied with a String-valued expression whose evaluation raises exception E2, then E2 is what gets propagated.

  RM references: 11.03 (4/2)

* ‘AI12-0065 Descendants of incomplete views (0000-00-00)’

  This AI is a clarification of potentially confusing wording. GNAT correctly handles the example given in AARM 7.3.1(5.b-5.d), which illustrates the topic of this AI.

  RM references: 7.03.01 (5.2/3)

* ‘AI12-0067 Accessibility level of explicitly aliased parameters of procedures and entries (0000-00-00)’

  The AI fixes a case where the intent was fairly obvious but the RM wording failed to mention a case (with the result that the accessibility level of an explicitly aliased parameter of a procedure or entry was undefined even though the intent was clear).

  RM references: 3.10.02 (7/3)

* ‘AI12-0068 Predicates and the current instance of a subtype (2020-05-06)’

  AI12-0068 is a binding interpretation that defines the current instance name in a type or subtype aspect to be a value rather than an object. This affects attributes whose prefix is a current instance in predicates, type invariants, and `Default_Initial_Condition` aspects. In particular, in the case of the `Constrained` attribute the value will always be True, and formerly legal attributes that require an object as their prefix (such as `Size`, `Access`, `Address`, etc.) are illegal when applied to a current instance in type and subtype aspects.

  RM references: 8.06 (17/3)

* ‘AI12-0069 Inconsistency in Tree container definition (0000-00-00)’

  The description of how iteration over a Tree container’s elements was contradictory in some cases regarding whether a cursor designating the Root node is included in the iteration. This contradiction was resolved. In the “!ACATS Test” section of the AI, it says that if an implementation were to get this wrong then almost any attempt to iterate over any tree would fail at runtime.

  RM references: A.18.10 (153/3) A.18.10 (155/3) A.18.10 (157/3) A.18.10 (159/3)

* ‘AI12-0070 9.3(2) does not work for anonymous access types (0000-00-00)’

  The RM contained some old wording about the master of an allocated object that only made sense for named access types. The AI clarifies the wording to clearly state the scope of validity and ensures that the paragraph does not contradict 3.10.2's rules for anonymous access types.

RM references: 3.10.02 (13.1/3) 9.03 (2)

* 'AI12-0071 Order of evaluation when multiple predicates apply (2015-08-10)'

  AI12-0071 specifies the semantics of multiple/inherited predicates on a single subtype.

  RM references: 3.02.04 (4/3) 3.02.04 (6/3) 3.02.04 (30/3) 3.02.04 (31/3) 3.02.04 (32/3) 3.02.04 (33/3) 3.02.04 (35/3) 3.05.05 (7.1/3) 3.05.05 (7.2/3) 3.05.05 (7.3/3) 3.08.01 (10.1/3) 3.08.01 (15/3) 4.05.02 (29/3) 4.05.02 (30/3) 4.06 (51/3) 4.09.01 (10/3) 5.04 (7/3) 5.05 (9/3) 13.09.02 (3/3) 13.09.02 (12)

* 'AI12-0072 Missing rules for Discard_Names aspect (0000-00-00)'

  Clarify that Discard_Names is an aspect, not just a pragma.

  RM references: C.05 (1) C.05 (5) C.05 (7/2) C.05 (8)

* 'AI12-0073 Synchronous Barriers are not allowed with Ravenscar (2020-02-24)'

  Ada 2022 adds (as a binding interpretation) a `No_Dependence => Ada.Synchronous_Barriers` restriction to the Ravenscar profile.

  RM references: D.13 (6/3)

* 'AI12-0074 View conversions and out parameters passed by copy (2020-03-26)'

  This Ada 2022 AI makes illegal some cases of out parameters whose type has a `Default_Value` aspect.

  RM references: 4.06 (56) 6.04.01 (6.25/3) 6.04.01 (13.1/3)

* 'AI12-0075 Static expression functions (2020-04-13)'

  Ada 2022 defines a new aspect `Static` that can be specified on expression functions. Such an expression function can be called in contexts requiring static expressions when the actual parameters are all static, allowing for greater abstraction in complex static expressions.

  RM references: 4.09 (21) 6.08 (3/4) 6.08 (5/4) 6.08 (6/4) 7.03.02 (8.2/5) 7.03.02 (15/4) 7.03.02 (16/4) 7.03.02 (17/4) 7.03.02 (19/4) 7.03.02 (20/5)

* 'AI12-0076 Variable state in pure packages (0000-00-00)'

  Defines an obscure constant-modifying construct to be erroneous. The issue is that the current instance of a type is a variable object, so the following is legal:

```
  type T;
  type T_Ref (Access_To_Variable : access T) is null record;
  type T is limited record
     Self : T_Ref (T'Access);
     Int : Integer;
  end record;

  Obj : constant T := (Self => <>, Int => 123);
begin
  Obj.Self.Access_To_Variable.Int := 456; -- modifying a component of a constant
```

  In cases where constancy is really needed (e.g., for an object declared in a Pure context), such a case needs to be erroneous.

  RM references: 10.02.01 (17/3) E.02.02 (17/2)

* 'AI12-0077 Has_Same_Storage on objects of size zero (2020-03-30)'

  This binding interpretation requires the Has_Same_Storage attribute to return always *false* for objects that have a size of zero.

  RM references: 13.03 (73.4/3)

* 'AI12-0078 Definition of node for tree container is confusing (0000-00-00)'

  Clarifies the expected behavior in processing tree containers.

  RM references: A.18.10 (2/3) A.18.10 (3/3)

* 'AI12-0081 Real-time aspects need to specify when they are evaluated (0000-00-00)'

  Clarify the point at which Priority and Interrupt_Priority aspect expressions are evaluated.

  RM references: D.01 (17/3) D.16 (9/3)

* 'AI12-0084 Box expressions in array aggregates (2014-12-15)'

  This AI addresses an issue where compiler used to fail to initialize components of a multidimensional aggregates with box initialization when scalar components have a specified default value. The AI clarifies that in an array aggregate with box (i.e., `<>`) component values, the `Default_Component_Value` of the array type (if any) should not be ignored.

  RM references: 4.03.03 (23.1/2)

* 'AI12-0085 Missing aspect cases for Remote_Types (0000-00-00)'

  A distributed systems annex (Annex E) clarification. Aspect specifications that are forbidden using attribute definition clause syntax are also forbidden using aspect_specification syntax.

  RM references: E.02.02 (17/2)

* 'AI12-0086 Aggregates and variant parts (2019-08-14)'

  In Ada 2012, a discriminant value that governs an active variant part in an aggregate had to be static. AI12-0086 relaxes this restriction: If the subtype of the discriminant value is a static subtype all of whose values select the same variant, then the expression for the discriminant is allowed to be nonstatic.

  RM references: 4.03.01 (17/3) 4.03.01 (19/3)

* 'AI12-0088 UTF_Encoding.Conversions and overlong characters on input (0000-00-00)'

  Clarify that overlong characters are acceptable on input even if we never generate them as output.

  RM references: A.04.11 (54/3) A.04.11 (55/3)

* 'AI12-0089 Accessibility rules need to take into account that a generic function is not a (0000-00-00)'

  Fix cases in RM wording where the accessibility rules for a function failed to take into account the fact that a generic function is not a function. For example, a generic function with an explicitly aliased parameter should be able to return references to that parameter in the same ways that a (non-generic) function can. The previous wording did not allow that.

  RM references: 3.10.02 (7/3) 3.10.02 (19.2/3) 3.10.02 (19.3/3) 6.05 (4/3)

* 'AI12-0093 Iterator with indefinite cursor (0000-00-00)'

  A clarification that confirms what GNAT is already doing.

  RM references: 5.05.02 (8/3) 5.05.02 (10/3)

* 'AI12-0094 An access_definition should be a declarative region (0000-00-00)'

  Fixes wording omission in the RM, confirming that the behaviour of GNAT is correct.

  RM references: 8.03 (2) 8.03 (26/3)

* 'AI12-0095 Generic formal types and constrained partial views (0000-00-00)'

  Deciding whether an actual parameter corresponding to an explicitly aliased formal parameter is legal depends on (among other things) whether the parameter type has a constrained partial view. The AI clarifies how this compile-time checking works in the case of a generic formal type (assume the best in the spec and recheck each instance, assume the worst in a generic body).

  RM references: 3.10.02 (27.2/3) 4.06 (24.16/2) 6.04.01 (6.2/3) 12.05.01 (15)

* 'AI12-0096 The exception raised when a subtype conversion fails a predicate check (0000-00-00)'

  Clarify that the Predicate_Failure aspect works the same in a subtype conversion as in any other context.

  RM references: 4.06 (57/3)

* 'AI12-0097 Tag of the return object of a simple return expression (0000-00-00)'

  Clarify wording about the tag of a function result in the case of a simple (i.e. not extended) return statement in a function with a class-wide result type.

  RM references: 6.05 (8/3)

* 'AI12-0098 Problematic examples for ATC (0000-00-00)'

  The AI clarifies reference manual examples, there is no compiler impact.

  RM references: 9.07.04 (13)

* 'AI12-0099 Wording problems with predicates (2020-05-04)'

  When extending a task or protected type from an ancestor interface subtype with a predicate, a link error can occur due to the compiler failing to generate the predicate-checking function. This AI clarifies the requirement for such predicate inheritance for concurrent types.

  RM references: 3.02.04 (4/4) 3.02.04 (12/3) 3.02.04 (20/3)

* 'AI12-0100 A qualified expression makes a predicate check (2020-02-17)'

  The compiler now enforces predicate checks on qualified expressions when the qualifying subtype imposes a predicate.

  RM references: 4.07 (4)

* 'AI12-0101 Incompatibility of hidden untagged record equality (2019-10-31)'

  AI12-0101 is a binding interpretation that removes a legality rule that prohibited the declaration of a primitive equality function for a private type in the private part of its enclosing package (either before or after the completion of the type) when the type is completed as an untagged record type. Such declarations are now accepted in Ada 2012 and later Ada versions.

As a consequence of this work, some cases where the implementation of AI05-0123 was incomplete were corrected. More specifically, if a user-defined equality operator is present for an untagged record type in an Ada 2012 program, that user-defined equality operator will be (correctly) executed in some difficult-to-characterize cases where the predefined component-by-component comparison was previously being (incorrectly) executed. This can arise, for example, in the case of the predefined equality operation for an enclosing composite type that has a component of the user-defined primitive equality op's operand type. This correction means that the impact of this change is not limited solely to code that was previously rejected at compile time.

RM references: 4.05.02 (9.8/3)

* 'AI12-0102 Stream_IO.File_Type has Preelaborable_Initialization (0000-00-00)'

  Modifies the declaration of one type in a predefined package. GNAT's version of `Ada.Streams.Stream_IO` already had this modification (the `Preelaborable_Initialization` pragma).

  RM references: A.12.01 (5)

* 'AI12-0103 Expression functions that are completions in package specifications (0000-00-00)'

  Clarifies that expression functions that are completions do not cause "general" freeze-everybody-in-sight freezing like a subprogram body.

  RM references: 13.14 (3/3) 13.14 (5/3)

* 'AI12-0104 Overriding an aspect is undefined (0000-00-00)'

  A clarification of the wording in RM, no compiler impact.

  RM references: 4.01.06 (4/3) 4.01.06 (17/3)

* 'AI12-0105 Pre and Post are not allowed on any subprogram completion (0000-00-00)'

  Language-defined aspects (e.g., `Post`) cannot be specified as part of the completion of a subprogram declaration. Fix a hole in the RM wording to clarify that this general rule applies even in the special cases where the completion is either an expression function or a null procedure.

  RM references: 13.01.01 (18/3)

* 'AI12-0106 Write'Class aspect (0000-00-00)'

  Clarify that the syntax used in an ACATS test BDD2005 for specifying a class-wide streaming aspect is correct.

  RM references: 13.01.01 (28/3) 13.13.02 (38/3)

* 'AI12-0107 A prefixed view of a By_Protected_Procedure interface has convention protected (2020-06-05)'

  A prefixed view of a subprogram with aspect Synchronization set to By_Protected_Procedure has convention protected.

  RM references: 6.03.01 (10.1/2) 6.03.01 (12) 6.03.01 (13)

* 'AI12-0109 Representation of untagged derived types (2019-11-12)'

  Ada disallows a nonconforming specification of a type-related representation aspect of an untagged by-reference type. The motivation for this rule is to ensure that a parent type and a later type derived from the parent agree with respect to such aspects. AI12-0109 disallows a construct that otherwise could be used to get around this rule: an aspect specification for the parent type that occurs after the declaration of the derived type.

RM references: 13.01 (10/3)

* 'AI12-0110 Tampering checks are performed first (2020-04-14)'

  AI12-0110 requires tampering checks in the containers library to be performed first, before any other checks.

  RM references: A.18.02 (97.1/3) A.18.03 (69.1/3) A.18.04 (15.1/3) A.18.07 (14.1/3) A.18.10 (90/3) A.18.18 (35/3)

* 'AI12-0112 Contracts for container operations (0000-00-00)'

  A representation change replacing english descriptions of contracts for operations on predefined container types with pre/post-conditions. No compiler impact.

  RM references: A.18.02 (99/3) 11.04.02 (23.1/3) 11.05 (23) 11.05 (26) A (4) A.18 (10)

* 'AI12-0114 Overlapping objects designated by access parameters are not thread-safe (0000-00-00)'

  There are rules saying that concurrent calls to predefined subprograms don't interfere with each other unless actual parameters overlap. The AI clarifies that such an interference is also possible if overlapping objects are reachable via access dereferencing from actual parameters of the two calls.

  RM references: A (3/2)

* 'AI12-0116 Private types and predicates (0000-00-00)'

  Clarify that the same aspect cannot be specified twice for the same type. `Dynamic_Predicate`, for example, can be specified on either the partial view of a type or on the completion in the private part, but not on both.

  RM references: 13.01 (9/3) 13.01 (9.1/3)

* 'AI12-0117 Restriction No_Tasks_Unassigned_To_CPU (2020-06-12)'

  This AI adds a restriction No_Tasks_Unassigned_To_CPU to provide safe use of Ravenscar.

  The CPU aspect is specified for the environment task. No CPU aspect is specified to be statically equal to `Not_A_Specific_CPU`. If aspect CPU is specified (dynamically) to the value `Not_A_Specific_CPU`, then Program_Error is raised. If Set_CPU or `Delay_Until_And_Set_CPU` are called with the CPU parameter equal to `Not_A_Specific_CPU`, then `Program_Error` is raised.

  RM references: D.07 (10.8/3)

* 'AI12-0120 Legality and exceptions of generalized loop iteration (0000-00-00)'

  Clarify that the expansion-based definition of generalized loop iteration includes legality checking. If the expansion would be illegal (for example, because of passing a constant actual parameter in a call when the mode of the corresponding formal parameter is in-out), then the loop is illegal too.

  RM references: 5.05.02 (6.1/4) 5.05.02 (10/3) 5.05.02 (13/3)

* 'AI12-0121 Stream-oriented aspects (0000-00-00)'

  Clarify that streaming-oriented aspects (e.g., Read) can be specified using aspect_specification syntax, not just via an attribute definition clause.

  RM references: 13.13.02 (38/3)

* 'AI12-0124 Add Object'Image (2017-03-24)'

  The corrigendum of Ada 2012 extends attribute `'Image following` the syntax for the GNAT `'Img` attribute. This AI fixes a gap in the earlier implementation, which did not recognize function calls and attributes that are functions as valid object prefixes.

  RM references: 3.05 (55/3)

* 'AI12-0125-3 Add @ as an abbreviation for the LHS of an assignment (2016-11-11)'

  This AI introduces the use of the character '@' as an abbreviation for the left-hand side of an assignment statement, usable anywhere within the expression on the right-hand side. To use this feature the compilation flag -gnat2022 must be specified.

  RM references: 5.02.01 (0) 2.02 (9) 3.03 (21.1/3) 4.01 (2/3) 8.06 (9/4)

* 'AI12-0127 Partial aggregate notation (2016-10-12)'

  This AI describes a new constructor for aggregates, in terms of an existing record or array object, and a series of component-wise modifications of its value, given by named associations for the modified components. To use this feature the compilation flag `-gnat2022` must be specified.

  RM references: 4.03 (2) 4.03 (3/2) 4.03 (4) 4.03.01 (9) 4.03.01 (15/3) 4.03.01 (16/4) 4.03.01 (17/5) 4.03.01 (17.1/2) 4.03.03 (4) 4.03.03 (14) 4.03.03 (17/5) 4.03.04 (0) 7.05 (2.6/2)

* 'AI12-0128 Exact size access to parts of composite atomic objects (2019-11-24)'

  According to this AI, the compiler generates full access to atomic composite objects even if the access is only partial in the source code. To use this feature the compilation flag `-gnat2022` must be specified.

  RM references: C.06 (13.2/3) C.06 (19) C.06 (20) C.06 (22/2) C.06 (25/4)

* 'AI12-0129 Make protected objects more protecting (2020-07-01)'

  A new aspect Exclusive_Functions has been added to the language to force the use of read/write locks on protected functions when needed.

  RM references: 9.05.01 (2) 9.05.01 (4) 9.05.01 (5) 9.05.01 (7) 9.05.03 (15) 9.05.03 (23)

* 'AI12-0130 All I/O packages should have Flush (2016-07-03)'

  The Flush routine has been added for the `Sequential_IO` and `Direct_IO` standard packages in the Ada 2012 COR.1:2016. The Flush routine here is equivalent to the one found in `Text_IO`. The `Flush` procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file.

  RM references: A.08.01 (10) A.08.02 (28/3) A.08.04 (10) A.10.03 (21) A.12.01 (28/2) A.12.01 (28.6/1)

* 'AI12-0131 Inherited Pre'Class when unspecified on initial subprogram (0000-00-00)'

  If T1 is a tagged type with a primitive P that has no class-wide precondition, and if T2 is an extension of T1 which overrides the inherited primitive P, then that overriding P is not allowed to have a class-wide precondition. Allowing it would be ineffective except in corner cases where it would be confusing.

  RM references: 6.01.01 (17/3) 6.01.01 (18/3)

* 'AI12-0132 Freezing of renames-as-body (2020-06-13)'

  This AI clarifies that a renames-as-body freezes the expression of any expression function that it renames.

RM references: 13.14 (5/3)

* 'AI12-0133 Type invariants and default initialized objects (0000-00-00)'

  Clarify that invariant checking for a default-initialized object is performed regardless of where the object is declared (in particular, even when the full view of the type is visible).

  RM references: 7.03.02 (10.3/3)

* 'AI12-0135 Enumeration types should be eligible for convention C (0000-00-00)'

  Ada previously allowed but did not require supporting specifying convention C for an enumeration type. Now it is required that an implementation shall support it.

  RM references: B.01 (14/3) B.01 (41/3) B.03 (65)

* 'AI12-0136 Language-defined packages and aspect Default_Storage_Pool (0000-00-00)'

  Clarify that the effect of specifying Default_Storage_Pool for an instance of a predefined generic is implementation-defined. No compiler impact.

  RM references: 13.11.03 (5/3)

* 'AI12-0137 Incomplete views and access to class-wide types (0000-00-00)'

  If the designated type of an access type is incomplete when the access type is declared, then we have rules about whether we get a complete view when a value of the access type is dereferenced. Clarify that analogous rules apply if the designated type is class-wide.

  RM references: 3.10.01 (2.1/2)

* 'AI12-0138 Iterators of formal derived types (2021-02-11)'

  AI12-0138 specifies the legality rules for confirming specifications of nonoverridable aspects. This completes the legality checks for aspect `Implicit_Dereference` and simplifies the checks for those aspects that are inherited operations.

  RM references: 13.01.01 (18/4) 13.01.01 (34/3) 4.01.05 (6/3) 4.01.06 (5/3) 4.01.06 (6/3) 4.01.06 (7/3) 4.01.06 (8/3) 4.01.06 (9/3) 5.05.01 (11/3)

* 'AI12-0140 Access to unconstrained partial view when full view is constrained (0000-00-00)'

  Clarify some confusion about about whether what matters when checking whether designated subtypes statically match is the view of the designated type that is currently available v.s. the view that was available when the access type was declared.

  RM references: 3.02 (7/2) 7.03.01 (5/1)

* 'AI12-0143 Using an entry index of a family in a precondition (2022-04-05)'

  Ada 2022 adds the `Index` attribute, which allows the use of the entry family index of an entry call within preconditions and post-conditions.

  RM references: 6.01.01 (30/3) 9.05.04 (5/3)

* 'AI12-0144 Make Discrete_Random more flexible (2020-01-31)'

  A new function Random with First/Last parameters is provided in the `Ada.Numerics.Discrete_Random` package.

  RM references: A.05.02 (20) A.05.02 (32) A.05.02 (41) A.05.02 (42)

* ‘AI12-0145 Pool_of_Subpool returns null when called too early (0000-00-00)’

  Clarify that if you ask for the pool of a subpool (by calling `Pool_Of_Subpool`) before `Set_Pool_of_Subpool` is called, then the result is null.

  RM references: 13.11.04 (20/3)

* ‘AI12-0147 Expression functions and null procedures can be declared in a protected_body (2015-03-05)’

  AI12-0147 specifies that null procedures and expression functions are now allowed in protected bodies.

  RM references: 9.04 (8/1)

* ‘AI12-0149 Type invariants are checked for functions returning access-to-type (0000-00-00)’

  Extend the rule saying that `Type_Invariant` checks are performed for access-to-T parameters (where T has a specified `Type_Invariant`) so that the rule also applies to function results.

  RM references: 7.03.02 (19.3/4)

* ‘AI12-0150 Class-wide type invariants and statically bound calls (0000-00-00)’

  The same approach used in AI12-0113 to ensure that contract-related calls associated with a call to a subprogram “match” with respect to dispatching also applies to `Type_Invariant` checking.

  RM references: 7.03.02 (3/3) 7.03.02 (5/3) 7.03.02 (9/3) 7.03.02 (22/3)

* ‘AI12-0154 Aspects of library units (0000-00-00)’

  Clarify that an aspect_specification for a library unit is equivalent to a corresponding aspect-specifying pragma.

  RM references: 13.01.01 (32/3)

* ‘AI12-0156 Use subtype_indication in generalized iterators (0000-00-00)’

  For iterating over an array, we already allow (but do not require) explicitly providing a subtype indication in an iterator_specification. Tee AI generalizes this to handle the case where the element type of the array is of an anonymous access type. This also allows (but does not require) explicitly naming the cursor subtype in a generalized iterator. The main motivation for allowing these new cases is improving readability by making it easy to infer the (sub)type of the iteration object just by looking at the loop.

  RM references: 5.05.02 (2/3) 5.05.02 (5/4) 5.05.02 (7/3) 3.10.02 (11.1/2)

* ‘AI12-0157 Missing rules for expression functions (0000-00-00)’

  Clarify that an expression function behaves like a single-return-statement function in more cases: it can return an aggregate without extra parens, the expression has an applicable index constraint, and the same accessibility rules apply in both cases.

  For instance, the code below is legal:

  ```
  subtype S is String (1 .. 10);
  function f return S is (others => '?');
  ```

  RM references: 3.10.02 (19.2/4) 3.10.02 (19.3/4) 4.03.03 (11/2) 6.08 (2/3) 6.08 (3/3) 6.08 (5/3) 6.08 (6/3) 6.08 (7/3) 7.05 (2.9/3) 13.14 (5.1/4) 13.14 (5.2/4) 13.14 (8/3) 13.14 (10.1/3) 13.14 (10.2/3) 13.14 (10.3/3)

* ‘AI12-0160 Adding an indexing aspect to an indexable container type (0000-00-00)’

  If the parent type of a derived type has exactly one of the two indexing aspects (that is, constant_indexing and variable_indexing) specified, then the derived type cannot have a specification for the other one.

  RM references: 4.01.06 (6/4) 4.01.06 (9/4) 3.06 (22.2/3)

* ‘AI12-0162 Memberships and Unchecked_Unions (0000-00-00)’

  Clarify that membership tests for unchecked_union types work consistently when testing membership in more than one subtype (X in AA | BB | CC) as when testing for one.

  RM references: B.03.03 (25/2)

* ‘AI12-0164 Max_Entry_Queue_Length aspect for entries (2019-06-11)’

  AI12-0164 defines pragma and aspect `Max_Entry_Queue_Length` in addition to the GNAT-specific equivalents `Max_Queue_Length` and `Max_Entry_Queue_Depth`.

  RM references: D.04 (16)

* ‘AI12-0165 Operations of class-wide types and formal abstract subprograms (2021-10-19)’

  Ada 2022 specifies that when the controlling type of a formal abstract subprogram declaration is a formal type, and the actual type is a class-wide type T’Class, the actual subprogram can be an implicitly declared subprogram corresponding to a primitive operation of type T.

  RM references: 12.06 (8.5/2)

* ‘AI12-0166 External calls to protected functions that appear to be internal calls (2016-11-15)’

  According to this AI, the compiler rejects a call to a protected operation when the call appears within a precondition for another protected operation.

  RM references: 6.01.01 (34/3) 9.05 (3/3) 9.05 (7.1/3)

* ‘AI12-0167 Type_Invariants and tagged-type View Conversions (0000-00-00)’

  This AI clarifies that no invariant check is performed in a case where an invariant-violating value is assigned to a component. This confirms the current compiler behavior.

  RM references: 7.03.02 (9/4)

* ‘AI12-0168 Freezing of generic instantiations of generics with bodies (0000-00-00)’

  Adjust freezing rules to be compatible with AI12-0103-1. The change confirms the current compiler behavior.

  RM references: 13.14 (3/4)

* ‘AI12-0169 Aspect specifications for entry bodies (0000-00-00)’

  Change syntax to allow aspect specifications for implementation-defined aspects on entry bodies. The change doesn’t influence any of the language-defined aspects and is solely required for SPARK.

  RM references: 9.05.02 (5)

* ‘AI12-0170 Abstract subprogram calls in class-wide precondition expressions (2020-07-06)’

This AI specifies rules for calls to abstract functions within class-wide preconditions and postconditions.

RM references: 3.09.03 (7) 6.01.01 (7/4) 6.01.01 (18/4) 6.01.01 (18.2/4)

* 'AI12-0172 Raise expressions in limited contexts (2019-07-29)'

  The compiler has been enhanced to support the use of raise expressions in limited contexts.

  RM references: 7.05 (2.1/3)

* 'AI12-0173 Expression of an extended return statement (0000-00-00)'

  Fix the wording related to expression of an extended return statement that was made ambiguous by changes of syntax in other AI's. No compiler changes involved.

  RM references: 6.05 (3/2) 6.05 (5/3)

* 'AI12-0174 Aggregates of Unchecked_Unions using named notation (0000-00-00)'

  In many cases, it is illegal to name a discriminant of an unchecked_union type. Relax this rule to allow the use of named notation in an aggregate of an unchecked_union type.

  RM references: B.03.03 (9/3)

* 'AI12-0175 Preelaborable packages with address clauses (2020-03-20)'

  The compiler nows accepts calls to certain functions that are essentially unchecked conversions in preelaborated library units. To use this feature the compilation flag `-gnat2022` must be specified.

  RM references: 10.02.01 (7)

* 'AI12-0179 Failure of postconditions of language-defined units (0000-00-00)'

  A clarification that expressing postconditions for predefined units via RM wording or via `Post` aspect specifications are equivalent. In particular, the expression in such a `Post` aspect specification should not yield False. No implementation changes needed.

  RM references: 1.01.03 (17/3) 11.04.02 (23.1/3)

* 'AI12-0180 Using protected subprograms and entries within an invariant (2020-06-22)'

  AI12-0180 makes entries and protected subprograms directly visible within Invariant aspects of a task or protected type.

  RM references: 13.01.01 (12/3)

* 'AI12-0181 Self-referencing representation aspects (0000-00-00)'

  Clarify that a name or expression which freezes an entity cannot occur in an aspect specification for that entity.

  RM references: 13.01 (9/4) 13.01 (9.1/4) 13.14 (19)

* 'AI12-0182 Pre'Class and protected operations (0000-00-00)'

  Confirm that Pre'Class and Post'Class cannot be specified for a protected operation. No language change.

  RM references: 13.01.01 (16/3)

* 'AI12-0184 Long Long C Data Types (2020-01-30)'

  Two new types `long_long` and `unsigned_long_long` are introduced in the package `Interfaces.C`.

  RM references: B.03 (71.3/3)

* 'AI12-0185 Resolution of postcondition-specific attributes (0000-00-00)'

  Clarify resolution rules for `'Old` and `'Result` attribute references to match original intent.

  RM references: 6.01.01 (7/4) 6.01.01 (8/3) 6.01.01 (26.10/4) 6.01.01 (29/3)

* 'AI12-0186 Profile freezing for the Access attribute (0000-00-00)'

  Clarify that the use of Some_Subprogram'Access does not freeze the profile of Some_Subprogram.

  RM references: 13.14 (15)

* 'AI12-0187 Stable properties of abstract data types (2020-11-04)'

  Ada 2022 defines a new aspect, `Stable_Properties`, for use in generating additional postcondition checks for subprograms.

  RM references: 7.03.04 (0) 13.01.01 (4/3)

* 'AI12-0191 Clarify "part" for type invariants (0000-00-00)'

  Clarify that for purposes of determining whether an invariant check is required for a "part" of an object, we do not look at "parts" which do not correspond to "parts" of the nominal type of the object. For example, if we have a parameter Param of a tagged type T1 (or equivalently of type T1'Class), and type T2 is an extension of T1 which declares a component Foo, and T1'Class (Param)'Tag = T2'Tag, then no invariant check is performed for Param's Foo component (or any subcomponent thereof).

  RM references: 3.03 (23/5) 3.09.01 (4.1/2) 6.08 (5.8/5) 7.03.02 (8.3/5) 7.03.02 (8.4/5) 7.03.02 (8.5/5) 7.03.02 (8.6/5) 7.03.02 (8.7/5) 7.03.02 (8.8/5) 7.03.02 (8.9/5) 7.03.02 (8.10/5) 7.03.02 (8.11/5) 7.03.02 (8.12/5) 7.03.02 (10.1/4) 7.03.02 (15/5) 7.03.02 (17/4) 7.03.02 (18/4) 7.03.02 (19/4) 13.13.02 (9/3)

* 'AI12-0192 "requires late initialization" and protected types (2020-03-11)'

  This AI clarifies that components of a protected type require late initialization when their initialization references (implicitly) the current instance of the type.

  RM references: 3.03.01 (8.1/2)

* 'AI12-0194 Language-defined aspects and entry bodies (0000-00-00)'

  The AI Includes entry bodies on the list of bodies for which no language-defined aspects can be specified (although specifying an implementation-defined aspect may be allowed).

  A wording change, no implementation impact.

  RM references: 13.01.01 (17/3)

* 'AI12-0195 Inheriting body but overriding precondition or postcondition (2021-08-11)'

  Ada 2022 specifies that if a primitive with a class-wide precondition or postcondition is inherited, and some primitive function called in the class-wide precondition or postcondition is overridden, then a dispatching call to the first primitive with a controlling operand that has the tag of the overriding type is required to check both the interpretation using the overriding function and the interpretation using the original overridden function.

  RM references: 6.01.01 (38/4)

* 'AI12-0196 Concurrent access to Ada container libraries (0000-00-00)'

  Clarify that parallel execution of operations which use cursors to refer to different elements of the same container does not violate the rules about erroneous concurrent access in some cases. That is, if C1 and C2 are cursors that refer to different elements of some container, then it is ok to concurrently execute an operation that is passed C1 and which accesses one element of the container, with another operation (perhaps the same operation, perhaps not) that is passed C2 and which accesses another element of the container.

  RM references: A.18 (2/2) A.18.02 (125/2) A.18.02 (133/3) A.18.02 (135/3) A.18.03 (81/3) A.18.04 (36/3) A.18.07 (34/2) A.18.10 (116/3)

* 'AI12-0198 Potentially unevaluated components of array aggregates (2020-05-13)'

  Ada 2022 enforces the detection of components that belong to a nonstatic or null range of index values of an array aggregate.

  RM references: 6.01.01 (22.1/4)

* 'AI12-0199 Abstract subprogram calls in class-wide invariant expressions (0000-00-00)'

  Class-wide type invariants do not apply to abstract types, to avoid various problems. Define the notion of a "corresponding expression" for a class-wide type invariant, replacing references to components as appropriate, taking into account rules for corresponding and specified discriminants when applying them to a nonabstract descendant.

  RM references: 7.03.02 (5/4) 7.03.02 (8/3)

* 'AI12-0201 Missing operations of static string types (2020-02-25)'

  Relational operators and type conversions of static string types are now static in Ada 2022.

  RM references: 4.09 (9) 4.09 (19) 4.09 (20) 4.09 (24)

* 'AI12-0203 Overriding a nonoverridable aspect (0000-00-00)'

  A corner case wording clarification that has no impact on compilers.

  RM references: 4.01.05 (5.1/4) 4.01.05 (7/3)

* 'AI12-0204 Renaming of a prefixed view (2020-02-24)'

  AI12-0204 clarifies that the prefix of a prefixed view that is renamed or passed as a formal subprogram must be renameable as an object.

  RM references: 8.05.04 (5.2/2) 12.06 (8.3/2) 4.01.03 (13.1/2) 4.01.06 (9/5)

* 'AI12-0205 Defaults for generic formal types (2021-04-01)'

  AI12-0205 specifies syntax and semantics that provide defaults for formal types of generic units. The legality rules guarantee that the default subtype_mark that is specified for a formal type would be a legal actual in any instantiation of the generic unit.

  RM references: 12.03 (7/3) 12.03 (10) 12.05 (2.1/3) 12.05 (2.2/3) 12.05 (7/2)

* 'AI12-0206 Nonoverridable should allow arbitrary kinds of aspects (0000-00-00)'

  A non-overridable aspect can have a value other than a name; for example, `Max_Entry_Queue_Length` is non-overridable and it has a scalar value. Part of adding support for `Max_Entry_Queue_Length` (which is already supported by GNAT).

  RM references: 13.01.01 (18.2/4) 13.01.01 (18.3/4) 13.01.01 (18.6/4)

* 'AI12-0207 Convention of anonymous access types (2020-02-01)'

  The convention of anonymous access elements of arrays now have the same convention as the array instead of convention Ada.

  RM references: 6.03.01 (13.1/3) B.01 (19) B.01 (21/3)
* 'AI12-0208 Predefined Big numbers support (0000-00-00)'

  Add predefined package `Ada.Numerics.Big_Numbers`.

  RM references: A.05.05 (0) A.05.06 (0) A.05.07 (0)
* 'AI12-0211 Interface types and inherited nonoverridable aspects (2020-08-24)'

  AI12-0211 introduces two new legality rules for Ada 2022. The first says that if a nonoverridable aspect is explicitly specified for a type that also inherits that aspect from another type (an ancestor or a progenitor), then the explicit aspect specification shall be confirming. The second says that if a type inherits a nonoverridable aspect from two different sources (this can only occur if at least one of the two is an interface type), then the two sources shall agree with respect to the given aspect. This AI is a binding interpretation, so these checks are performed even for earlier Ada versions. Because of compatibility concerns, an escape mechanism for suppressing these legality checks is provided: these new checks always pass if the `-gnatd.M` switch (relaxed RM semantics) is specified.

  RM references: 13.01.01 (18.3/5) 13.01.01 (18.4/4)
* 'AI12-0212 Container aggregates; generalized array aggregates (0000-00-00)'

  The AI defines a new feature: generalized array aggregates that already exists in GNAT.

  RM references: 4.03.05 (0) 1.01.04 (12) 1.01.04 (13) 2.01 (15) 2.02 (9/5) 3.07.01 (3) 3.08.01 (4) 4.03 (2/5) 4.03 (3/5) 4.03.01 (5) 4.03.03 (3/2) 4.03.03 (4/5) 4.03.03 (5.1/5) 4.03.03 (9) 4.03.03 (17/5) 4.03.03 (21) 4.03.03 (23.2/5) 4.03.03 (26) 4.03.03 (27) 4.03.03 (31) 4.03.04 (4/5) 4.04 (3.1/3) 11.02 (3) 13.01.01 (5/3) 13.01.01 (7/3) A.18.02 (8/3) A.18.02 (14/2) A.18.02 (47/2) A.18.02 (175/2) A.18.03 (6/3) A.18.05 (3/3) A.18.06 (4/3) A.18.08 (3/3) A.18.09 (4/3)
* 'AI12-0216 6.4.1(6.16-17/3) should never apply to composite objects (0000-00-00)'

  Fix wording so that parameter passing cases where there isn't really any aliasing problems or evaluation order dependency are classified as acceptable.

  No compiler impact.

  RM references: 6.04.01 (6.17/3)
* 'AI12-0217 Rules regarding restrictions on the use of the Old attribute are too strict (2020-03-25)'

  AI12-0217 loosens the rules regarding what is allowed as the prefix of a 'Old attribute reference. In particular, a prefix is now only required to "statically name" (as opposed to the previous "statically denote") an object. This means that components of composite objects that previously would have been illegal are now legal prefixes.

  RM references: 6.01.01 (24/3) 6.01.01 (27/3)
* 'AI12-0220 Pre/Post for access-to-subprogram types (2020-04-14)'

  Contract aspects can now be specified for access-to-subprogram types, as defined for Ada 2022 in this AI.

RM references: 6.01.01 (1/4) 6.01.01 (2/3) 6.01.01 (4/3) 6.01.01 (19/3) 6.01.01 (28/3) 6.01.01 (29/3) 6.01.01 (39/3) 13.01.01 (12/5)

* 'AI12-0222 Representation aspects and private types (0000-00-00)'

  Clarify that the rule against specifying a representation aspect for a type before the type is completely defined also applies in the case where aspect_specification syntax is used (not just in the case where a pragma or some other kind of representation item is used).

  GNAT already implements this.

  RM references: 13.01 (9/5) 13.01 (9.1/4) 13.01 (9.2/5)

* 'AI12-0225 Prefix of Obj'Image (0000-00-00)'

  Clarify some Object vs. Value corner cases to allow names that do not denote objects in more contexts, such as a qualified expression as a prefix of an Image attribute.

  RM references: 3.05 (55.1/4)

* 'AI12-0226 Make objects more consistent (0000-00-00)'

  Allow value conversions as objects. For instance this example becomes legal: `Long_Integer (Duration'Last)'Image`.

  RM references: 3.03 (11.1/3) 3.03 (21.1/3) 3.03 (23.8/5) 4.06 (58.1/4) 4.06 (58.3/4)

* 'AI12-0227 Evaluation of nonstatic universal expressions when no operators are involved (0000-00-00)'

  Nonstatic universal integer expressions are always evaluated at runtime as values of type root_integer; similarly, nonstatic universal real expressions are always evaluated at runtime as values of type root_real. This AI corrects a wording oversight. Previously, the above was only true if a call to operator was involved. With this change it is true in all cases.

  No compiler impact.

  RM references: 4.04 (10) 8.06 (29)

* 'AI12-0228 Properties of qualified expressions used as names (2020-02-19)'

  This AI clarifies that properties of a qualified object pass through a qualified expression used as a name. Specifically, "aliased" and "known to be constrained" are not changed by a qualified expression.

  RM references: 3.03 (23.7/3) 3.10 (9/3)

* 'AI12-0231 Null_Task_Id and Activation_Is_Complete (0000-00-00)'

  Add `Activation_Is_Complete` to the list of functions that raise P_E if passed `Null_Task_Id`, correcting an oversight.

  RM references: C.07.01 (15)

* 'AI12-0232 Rules for pure generic bodies (0000-00-00)'

  Clarify the rules for a generic body nested in a pure library unit.

  RM references: 10.02.01 (9/3) 10.02.01 (15.1/3) 10.02.01 (15.5/3)

* 'AI12-0233 Pre'Class for hidden operations of private types (0000-00-00)'

  Clarify how `Pre'Class` checking interacts with private-part overriding of inherited subprograms. A class-wide precondition can be checked at runtime even if it is specified in a private part that the caller cannot see into.

RM references: 6.01.01 (38/4)

* 'AI12-0234 Compare-and-swap for atomic objects (0000-00-00)'

  New predefined units for atomic operations (`System.Atomic_Operations` and child units thereof).

  RM references: C.06.01 (0) C.06.02 (0)

* 'AI12-0235 System.Storage_Pools should be pure (0000-00-00)'

  Change the predefined package System.Storage_Pools from preelaborated to pure.

  RM references: 13.11 (5)

* 'AI12-0236 declare expressions (2020-04-08)'

  A `declare expression` allows constant objects and renamings to be declared within an expression.

  RM references: 2.08 (6) 3.09.02 (3) 3.10.02 (9.1/3) 3.10.02 (16.1/3) 3.10.02 (32.2/3) 4.03.02 (5.4/3) 4.03.03 (15.1/3) 4.04 (7/3) 4.05.09 (0) 6.02 (10/4) 7.05 (2.1/5) 8.01 (2.1/4)

* 'AI12-0237 Getting the representation of an enumeration value (2020-01-31)'

  The GNAT-specific attributes `Enum_Rep` and `Enum_Val` have been standardized and are now also supported as Ada 2022 attributes.

  RM references: 13.04 (10) 13.04 (11/3)

* 'AI12-0242 Shorthand Reduction Expressions for Objects (0000-00-00)'

  Allow reduction expressions to iterate over an an array or an iterable object without having to explicitly create a value sequence.

  This allows, for instance, writing `A'Reduce("+", 0)` instead of the equivalent (but more verbose) `[for Value of A => Value]'Reduce("+", 0);`.

  RM references: 4.05.10 (0) 4.01.04 (6)

* 'AI12-0247 Potentially Blocking goes too far for Detect_Blocking (0000-00-00)'

  During a protected action, a call on a subprogram that contains a potentially blocking operation is considered a bounded error (so raising P_E is optional). This rule imposed an unreasonable implementation burden. The new rule introduced by this AI allows ignoring (i.e., not detecting) the problem until execution of a potentially blocking operation is actually attempted.

  RM references: 9.05 (55/5) 9.05 (56/5) 9.05.01 (18/5) H.05 (5/2)

* 'AI12-0249 User-defined numeric literals (2020-04-07)'

  Compiler support is added for three new aspects (`Integer_Literal`, `Real_Literal`, and `String_Literal`) as described in AI12-0249 (for `Integer_Literal` and `Real_Literal`), AI12-0295 (for `String_Literal`), and in two follow-up AIs (AI12-0325 and AI12-0342). For pre-Ada 2022 versions of Ada, these are treated as implementation-defined aspects. Some implementation work remains, particularly in the interactions between these aspects and tagged types.

  RM references: 4.02 (9) 4.02.01 (0) 4.09 (3)

* 'AI12-0250 Iterator Filters (2020-05-19)'

  This AI defines Ada 2022 feature of iterator filters, which can be applied to loop parameter specifications and iterator specifications.

RM references: 4.03.03 (21) 4.03.03 (26) 4.03.03 (31) 4.03.05 (0) 4.05.10 (0) 5.05 (4) 5.05 (7) 5.05 (9/4) 5.05 (9.1/4) 5.05 (10) 5.05.02 (2/3) 5.05.02 (10/3) 5.05.02 (11/3)

* 'AI12-0252 Duplicate interrupt handlers under Ravenscar (2018-07-05)'

  Ada Issue AI12-0252 requires that the runtime shall terminate with a Program_Error when more than one interrupt handler is attached to the same interrupt and the restriction No_Dynamic_Attachment is in effect.

  RM references: C.03.01 (13)

* 'AI12-0254 Bounded_Indefinite_Holders (2025-12-15)'

  This AI defines a new package `Ada.Containers.Indefinite_Holders` which is fully implemented by GNAT.

  RM references: A.18.32 (0)

* 'AI12-0256 Aspect No_Controlled_Parts (2021-01-26)'

  The compiler now supports the Ada 2022 aspect No_Controlled_Parts (see AI12-0256). When specified for a type, this aspect requires that the type and any of its ancestors must not have any controlled parts.

  RM references: H.04.01 (0) 13.01.01 (18.7/5)

* 'AI12-0258 Containers and controlled element types (0000-00-00)'

  Most predefined containers are allowed to defer finalization of container elements until the finalization of the container. This allows implementation flexibility but causes problems in some cases. AI12-0258 tightens up the rules for the indefinite containers to say that finalization happens earlier - if a client needs the tighter finalization guarantees, then it can use the indefinite containers (even if the element subtype in question is definite). Other solutions involving the holder generic are also possible.

  GNAT implements these tighter element finalization requirements for instances of the indefinite container generics.

  RM references: A.18 (10/4)

* 'AI12-0259 Lower bound of strings returned from Ada.Command_Line (0000-00-00)'

  Specify that the low-bound of a couple of predefined String-valued functions will always be one.

  RM references: A.15 (14) A.15 (16/3)

* 'AI12-0260 Functions Is_Basic and To_Basic in Wide_Characters.Handling (2020-04-01)'

  AI12-0260 is implemented for Ada 2022, providing the new functions `Is_Basic` and `To_Basic` in package `Ada.Wide_Characters.Handling`.

  RM references: 1.02 (8/3) A.03.05 (8/3) A.03.05 (20/3) A.03.05 (21/3) A.03.05 (33/3) A.03.05 (61/3)

* 'AI12-0261 Conflict in "private with" rules (0000-00-00)'

  If a library unit is only visible at some point because of a "private with", there are legality rules about a name denoting that entity. The AI cleans up the wording so that it captures the intent in a corner case involving a private-child library-unit subprogram. The previous wording incorrectly caused this case to be illegal.

  RM references: 10.01.02 (12/3) 10.01.02 (13/2) 10.01.02 (14/2) 10.01.02 (15/2) 10.01.02 (16/2)

* 'AI12-0262 Map-Reduce attribute (0000-00-00)'

  The AI defines Reduction Expressions to allow the programmer to apply the Map-Reduce paradigm to map/transform a set of values to a new set of values, and then summarize/reduce the transformed values into a single result value.

  RM references: 4.01.04 (1) 4.01.04 (6) 4.01.04 (11) 4.05.10 (0)
* 'AI12-0263 Update references to ISO/IEC 10646 (0000-00-00)'

  Change RM references to ISO/IEC 10646:2011 to instead refer to ISO/IEC 10646:2017. No compiler impact.

  RM references: 1.01.04 (14.2/3) 2.01 (1/3) 2.01 (3.1/3) 2.01 (4/3) 2.01 (4.1/5) 2.01 (5/3) 2.01 (15/3) 2.01 (4.1/5) 2.01 (5/3) 2.03 (4.1/5) 2.03 (5/3) 3.05.02 (2/3) 3.05.02 (3/3) 3.05.02 (4/3) A.01 (36.1/3) A.01 (36.2/3) A.03.02 (32.6/5) A.03.05 (51.2/5) A.03.05 (55/3) A.03.05 (59/3) A.04.10 (3/3) B.05 (21/5)
* 'AI12-0264 Overshifting and overrotating (0000-00-00)'

  Clarify Shift and Rotate op behavior with large shift/rotate amounts.

  RM references: B.02 (9)
* 'AI12-0265 Default_Initial_Condition for types (2020-11-13)'

  The aspect `Default_Initial_Condition`, originally proposed by SPARK and supported in GNAT, is now also included in Ada 2022. One change from the original implementation is that when the aspect is specified on ancestor types of a derived type, the ancestors' check expressions also apply to the derived type. `Default_Initial_Condition` checks are also now applied in cases of default initialization of components, allocators, ancestor parts of extension aggregates, and box associations of aggregates.

  RM references: 7.03.03 (0) 1.01.03 (17.1/5) 11.04.02 (23.2/5) 11.04.02 (23.3/5)
* 'AI12-0269 Aspect No_Return for functions reprise (2020-03-19)'

  This amendment has been implemented under the `-gnat2022` switch, and the compiler now accepts the aspect/pragma No_Return for functions and generic functions.

  RM references: 6.05.01 (0) 6.05.01 (1/3) 6.05.01 (3.1/3) 6.05.01 (3.4/3) 6.05.01 (5/2) 6.05.01 (6/2) 6.05.01 (7/2) J.15.02 (2/3) J.15.02 (3/3) J.15.02 (4/3)
* 'AI12-0272 (part 1) Pre/Postconditions for formal subprograms (0000-00-00)'

  Pre and Post aspects can be specified for a generic formal subprogram. `Default_Initial_Condition` can be specified for a generic formal private type.

  GNAT implements this with an exception of the part related to `Default_Initial_Condition`.

  RM references: 6.01.01 (1/5) 6.01.01 (39/5) 7.03.03 (1/5) 7.03.03 (2/5) 7.03.03 (8/5) 7.03.04 (5/5) F.01 (1)
* 'AI12-0275 Make subtype_mark optional in object renames (2020-01-28)'

  AI12-0275 allows object renamings to be declared without an explicit subtype_mark or access_definition. This feature can be used by compiling with the switch `-gnat2022`.

  RM references: 8.05.01 (2/3) 8.05.01 (3/2)
* 'AI12-0277 The meaning of "accessibility level of the body of F" (2020-09-09)'

  Clarify that the only time that an explicitly aliased formal parameter has different accessibility properties than an aliased part of a "normal" parameter is for the accessibility checking associated with a return statement.

RM references: 3.10.02 (19.2/4)

* ‘AI12-0278 Implicit conversions of anonymous return types (0000-00-00)’

  If a call to a function with an anonymous-access-type result is converted to a named access type, it doesn’t matter whether the conversion is implicit or explicit. the AI fixes hole where the previous rules didn’t cover the implicit conversion case.

  RM references: 3.10.02 (10.3/3)

* ‘AI12-0279 Nonpreemptive dispatching needs more dispatching points (2020-04-17)’

  Ada 2022 defines a new aspect *Yield* that can be specified in the declaration of a non-instance subprogram (including a generic formal subprogram), a generic subprogram, or an entry, to ensure that the associated subprogram has at least one task dispatching point during each invocation.

  RM references: D.02.01 (1.5/2) D.02.01 (7/5)

* ‘AI12-0280-2 Making ‘Old more flexible (2020-07-24)’

  For Ada 2022, AI12-0280-2 relaxes Ada’s restrictions on ‘Old attribute references whose attribute prefix does not statically name an entity. Previously, it was required that such an attribute reference must be unconditionally evaluated when the postcondition is evaluated; with the new rule, conditional evaluation is permitted if the relevant conditions can be evaluated upon entry to the subprogram with the same results as evaluation at the time of the postcondition’s evaluation. In this case, the ‘Old attribute prefix is evaluated conditionally (more specifically, the prefix is evaluated only if the result of that evaluation is going to be referenced later when the postcondition is evaluated).

  RM references: 6.01.01 (20/3) 6.01.01 (21/3) 6.01.01 (22/3) 6.01.01 (22.1/4) 6.01.01 (22.2/5) 6.01.01 (23/3) 6.01.01 (24/3) 6.01.01 (26/4) 6.01.01 (27/5) 6.01.01 (39/5)

* ‘AI12-0282 Atomic, Volatile, and Independent generic formal types (0000-00-00)’

  The AI specifies that the aspects `Atomic`, `Volatile`, `Independent`, `Atomic_Components`, `Volatile_Components`, and `Independent_Components` are specifiable for generic formal types. The actual type must have a matching specification.

  RM references: C.06 (6.1/3) C.06 (6.3/3) C.06 (6.5/3) C.06 (6.8/3) C.06 (12/3) C.06 (12.1/3) C.06 (21/4)

* ‘AI12-0285 Syntax for Stable_Properties aspects (0000-00-00)’

  The AI establishes the required named notation for a Stable_Properties aspect specification in order to avoid syntactic ambiguities.

  With the old syntax, an example like

  ```
  type Ugh is ...
     with Stable_Properties =\> Foo, Bar, Nonblocking, Pack;
  ```

  was problematic; `Nonblocking` and `Pack` are other aspects, while `Foo` and `Bar` are `Stable_Properties` functions. With the clarified syntax, the example above shall be written as:

  ```
  type Ugh is ...
    with Stable_Properties => (Foo, Bar), Nonblocking, Pack;
  ```

  RM references: 7.03.04 (2/5) 7.03.04 (3/5) 7.03.04 (4/5) 7.03.04 (6/5) 7.03.04 (7/5) 7.03.04 (9/5) 7.03.04 (10/5) 7.03.04 (14/5) 13.01.01 (4/5)

* 'AI12-0287 Legality Rules for null exclusions in renaming are too fierce (2020-02-17)'

  The null exclusion legality rules for generic formal object matching and object renaming now only apply to generic formal objects with mode in out.

  RM references: 8.05.01 (4.4/2) 8.05.01 (4.5/2) 8.05.01 (4.6/2) 8.05.04 (4.2/2) 12.04 (8.3/2) 12.04 (8.4/2) 12.04 (8.5/2) 12.04 (8.2/5) 12.06 (8.2/5)

* 'AI12-0289 Implicitly null excluding anonymous access types and conformance (2020-06-09)'

  AI12-0289 is implemented for Ada 2022, allowing safer use of access parameters when the partial view of the designated type is untagged, but the full view is tagged.

  RM references: 3.10 (26)

* 'AI12-0290 Restriction Pure_Barriers (2020-02-18)'

  The GNAT implementation of the Pure_Barriers restriction has been updated to match the Ada RM's definition as specified in this AI. Some constructs that were accepted by the previous implementation are now rejected, and vice versa. In particular, the use of a component of a component of a protected record in a barrier expression, as in "when Some_Component.Another_Component =>", formerly was (at least in some cases) not considered to be a violation of the Pure_Barriers restriction; that is no longer the case.

  RM references: D.07 (2) D.07 (10.10/4)

* 'AI12-0291 Jorvik Profile (2020-02-19)'

  The Jorvik profile is now implemented, as defined in this AI. For Ada 2012 and earlier versions of Ada, Jorvik is an implementation-defined profile whose definition matches its Ada 2022 definition.

  RM references: D.13 (0) D.13 (1/3) D.13 (4/3) D.13 (6/4) D.13 (9/3) D.13 (10/3) D.13 (11/4) D.13 (12/4)

* 'AI12-0293 Add predefined FIFO_Streams packages (0000-00-00)'

  The AI adds `Ada.Streams.Storage` and its two subunits `Bounded` and `Unbounded`.

  RM references: 13.13.01 (1) 13.13.01 (9) 13.13.01 (9.1/1)

* 'AI12-0295 User-defined string (2020-04-07)'

  Compiler support is added for three new aspects (`Integer_Literal`, `Real_Literal`, and `String_Literal`) as described in AI12-0249 (for `Integer_Literal` and `Real_Literal`), AI12-0295 (for `String_Literal`), and in two follow-up AIs (AI12-0325 and AI12-0342). For pre-Ada 2022 versions of Ada, these are treated as implementation-defined aspects. Some implementation work remains, particularly in the interactions between these aspects and tagged types.

  RM references: 4.02 (6) 4.02 (10) 4.02 (11) 3.06.03 (1) 4.02.01 (0) 4.09 (26/3)

* 'AI12-0301 Predicates should be checked like constraints for types with Default_Value (2020-02-25)'

  This AI clarifies that predicate checks apply for objects that are initialized by default and that are of a type that has any components whose subtypes specify `Default_Value` or `Default_Component_Value`.

  RM references: 3.02.04 (31/4)

* 'AI12-0304 Image attributes of language-defined types (2020-07-07)'

  According to this AI, `Put_Image` (and therefore `'Image`) is provided for the containers and for unbounded strings.

  RM references: 4.10 (0)

* 'AI12-0306 Split null array aggregates from positional array aggregates (0000-00-00)'

  The AI clarifies the wording of the references RM paragraphs without introducing any language changes.

  RM references: 4.03.03 (2) 4.03.03 (3/2) 4.03.03 (9/5) 4.03.03 (26/5) 4.03.03 (26.1/5) 4.03.03 (33/3) 4.03.03 (38) 4.03.03 (39) 4.03.03 (42)

* 'AI12-0307 Resolution of aggregates (2020-08-13)'

  The proposed new syntax for aggregates in Ada 2022 uses square brackets as delimiters, and in particular allows `[]` as a notation for empty array and container aggregates. This syntax is currently available as an experimental feature under the `-gnatX` flag.

  RM references: 4.03 (3/5)

* 'AI12-0309 Missing checks for pragma Suppress (0000-00-00)'

  The AI includes some previously overlooked run-time checks in the list of checks that are potentially suppressed via a pragma `Suppress`. For example, AI12-0251-1 adds a check that the number of chunks in a chunk_specification is not zero or negative. Clarify that suppressing `Program_Error_Check` suppresses that check too.

  RM references: 11.05 (10) 11.05 (19) 11.05 (20) 11.05 (22) 11.05 (24)

* 'AI12-0311 Suppressing client-side assertions for language-defined units (0000-00-00)'

  The AI defines some new assertion policies that can be given as arguments in a Suppress pragma (e.g., Calendar_Assertion_Check). GNAT recognizes and ignores those new policies, the checks are not implemented.

  RM references: 11.04.02 (23.5/5) 11.05 (23) 11.05 (26)

* 'AI12-0315 Image Attributes subclause improvements (0000-00-00)'

  Clarify that a named number or similar can be the prefix of an Image attribute reference.

  RM references: 4.10 (0)

* 'AI12-0318 No_IO should apply to Ada.Directories (2020-01-31)'

  The restriction No_IO now applies to and prevents the use of the `Ada.Directories package`.

  RM references: H.04 (20/2) H.04 (24/3)

* 'AI12-0321 Support for Arithmetic Atomic Operations and Test and Set (0000-00-00)'

  The AI adds some predefined atomic operations, e.g. package System.“Atomic_Operations.Test_And_Set“.

  RM references: C.06.03 (0) C.06.04 (0)

* 'AI12-0325 Various issues with user-defined literals (2020-04-07)'

  Compiler support is added for three new aspects (`Integer_Literal`, `Real_Literal`, and `String_Literal`) as described in AI12-0249 (for `Integer_Literal` and `Real_Literal`), AI12-0295 (for `String_Literal`), and in two follow-up AIs (AI12-0325 and

AI12-0342). For pre-Ada 2022 versions of Ada, these are treated as implementation-defined aspects. Some implementation work remains, particularly in the interactions between these aspects and tagged types.

RM references: 4.02 (6) 4.02 (10) 4.02 (11) 4.02.01 (0)

* 'AI12-0329 Naming of FIFO_Streams packages (0000-00-00)'

  The AI changes the name of predefined package `Ada.Streams.FIFO_Streams` to `Ada.Streams.Storage`.

  RM references: 13.13.01 (9/5) 13.13.01 (9.1/5)

* 'AI12-0331 Order of finalization of a subpool (0000-00-00)'

  Clarify that when a subpool is being finalized, objects allocated from that subpool are finalized before (not after) they cease to exist (i.e. object's storage has been reclaimed).

  RM references: 13.11.05 (5/3) 13.11.05 (6/3) 13.11.05 (7/3) 13.11.05 (7.1/4) 13.11.05 (8/3) 13.11.05 (9/3)

* 'AI12-0333 Predicate checks on out parameters (0000-00-00)'

  If a view conversion is passed as an actual parameter corresponding to an out-mode formal parameter, and if the subtype of the formal parameter has a predicate, then no predicate check associated with the conversion is performed.

  RM references: 3.02.04 (31/5) 4.06 (51/4) 6.04.01 (14)

* 'AI12-0335 Dynamic accessibility check needed for some requeue targets (0000-00-00)'

  Define a new runtime accessibility check for a corner case involving requeue statements.

  RM references: 9.05.04 (7/4)

* 'AI12-0336 Meaning of Time_Offset (0000-00-00)'

  The AI introduces changes to the predefined package `Ada.Calendar.Time_Zones`.

  RM references: 9.06.01 (6/2) 9.06.01 (35/2) 9.06.01 (40/2) 9.06.01 (41/2) 9.06.01 (42/3) 9.06.01 (90/2) 9.06.01 (91/2)

* 'AI12-0337 Simple_Name("/") in Ada.Directories (0000-00-00)'

  Clarify behavior of subprograms in the predefined package `Ada.Directories`. In particular, Simple_Name ("/") should return "/" on Unix-like systems.

  RM references: A.16 (47/2) A.16 (74/2) A.16 (82/3)

* 'AI12-0338 Type invariant checking and incomplete types (0000-00-00)'

  Clarify that type invariants for type T are not checked for incomplete types whose completion is not available, even if that completion has components of type T.

  RM references: 7.03.02 (20/5)

* 'AI12-0339 Empty function for Container aggregates (2020-08-06)'

  To provide uniform support for container aggregates, all standard container libraries have been enhanced with a function Empty, to be used when initializing an aggregate prior to inserting the specified elements in the object being constructed. All products have been updated to remove the ambiguities that may have arisen from previous uses of entities named Empty in our sources, and the expansion of container aggregates uses Empty wherever needed.

  RM references: A.18.02 (8/5) A.18.02 (12.3/5) A.18.02 (78.2/5) A.18.02 (98.6/5) A.18.03 (6/5) A.18.03 (10.2/5) A.18.03 (50.2/5) A.18.05 (3/5) A.18.05 (7.2/5) A.18.05

(37.3/5) A.18.05 (46/2) A.18.06 (4/5) A.18.06 (8.2/5) A.18.06 (51.4/5) A.18.08 (3/5) A.18.08 (8.1/5) A.18.08 (59.2/5) A.18.08 (68/2) A.18.09 (4/5) A.18.09 (9.1/5) A.18.09 (74.2/5) A.18.10 (15.2/5) A.18.18 (8.1/5) A.18.19 (6.1/5) A.18.20 (6/3) A.18.21 (6/3) A.18.22 (6/3) A.18.23 (6/3) A.18.24 (6/3) A.18.25 (8/3)

* 'AI12-0340 Put_Image should use a Text_Buffer (0000-00-00)'

  Add a new predefined package Ada.Strings.Text_Buffers (along with child units) and change the definition of Put_Image attribute to refer to it.

  RM references: A.04.12 (0) 4.10 (3.1/5) 4.10 (3.2/5) 4.10 (6/5) 4.10 (25.2/5) 4.10 (28/5) 4.10 (31/5) 4.10 (41/5) H.04 (23.2/5) H.04 (23.11/5)

* 'AI12-0342 Various issues with user-defined literals (part 2) (2020-04-07)'

  Compiler support is added for three new aspects (`Integer_Literal`, `Real_Literal`, and `String_Literal`) as described in AI12-0249 (for `Integer_Literal` and `Real_Literal`), AI12-0295 (for `String_Literal`), and in two follow-up AIs (AI12-0325 and AI12-0342). For pre-Ada 2022 versions of Ada, these are treated as implementation-defined aspects. Some implementation work remains, particularly in the interactions between these aspects and tagged types.

  RM references: 4.02.01 (0) 3.09.02 (1/2) 6.03.01 (22)

* 'AI12-0343 Return Statement Checks (2020-04-02)'

  This binding interpretation has been implemented and the accessibility, predicate, and tag checks prescribed by RM 6.5 are now performed at the appropriate points, as required by this AI.

  RM references: 6.05 (5.12/5) 6.05 (8/4) 6.05 (8.1/3) 6.05 (21/3)

* 'AI12-0345 Dynamic accessibility of explicitly aliased parameters (2020-09-09)'

  Further clarify (after AI12-0277) accessibility rules for explicitly aliased parameters.

  RM references: 3.10.02 (5) 3.10.02 (7/4) 3.10.02 (10.5/3) 3.10.02 (13.4/4) 3.10.02 (19.2/5) 3.10.02 (21)

* 'AI12-0350 Swap for Indefinite_Holders (2025-12-15)'

  Package `Ada.Containers.Indefinite_Holders` is implemented in GNAT, comprising the support for `Swap` as specified by this AI.

  RM references: A.18.18 (22/5) A.18.18 (67/5) A.18.18 (73/3) A.18.32 (13/5)

* 'AI12-0351 Matching for actuals for formal derived types (2020-04-03)'

  This binding interpretation requires the compiler to checks that an actual subtype in a generic parameter association of an instantiation is statically compatible (even when the actual is unconstrained) with the ancestor of an associated nondiscriminated generic formal derived type.

  RM references: 12.05.01 (7) 12.05.01 (8)

* 'AI12-0352 Early derivation and equality of untagged types (2020-07-09)'

  AI12-0352 clarifies that declaring a user-defined primitive equality operation for a record type T is illegal if it occurs after a type has been derived from T.

  RM references: 4.05.02 (9.8/4)

* 'AI12-0356 Root_Storage_Pool_With_Subpools should have Preelaborable_Initialization (0000-00-00)'

Add Preelaborable_Initialization pragmas for predefined types `Root_Storage_Pool_With_Subpools` and `Root_Subpool`.

RM references: 13.11.04 (4/3) 13.11.04 (5/3)

* 'AI12-0363 Fixes for Atomic and Volatile (2020-09-08)'

  This amendment has been implemented under the `-gnat2022` switch and the compiler now supports the `Full_Access_Only` aspect, which is mostly equivalent to GNAT's `Volatile_Full_Access`.

  RM references: 3.10.02 (26/3) 9.10 (1/5) C.06 (6.4/3) C.06 (6.10/3) C.06 (8.1/4) C.06 (12/5) C.06 (12.1/5) C.06 (13.3/5) C.06 (19.1/5)

* 'AI12-0364 Add a modular atomic arithmetic package (0000-00-00)'

  Generalize support for atomic integer operations to extend to modular types. Add new predefined generic package, `System.Atomic_Operations.Modular_Arithmetic`.

  RM references: C.06.05 (0) C.06.04 (1/5) C.06.04 (2/5) C.06.04 (3/5) C.06.04 (9/5)

* 'AI12-0366 Changes to Big_Integer and Big_Real (0000-00-00)'

  Simplify `Big_Integer ``and ``Big_Real` specs by eliminating explicit support for creating "invalid" values. No more `Optional_Big_[Integer,Real]` types.

  RM references: A.05.06 (0) A.05.07 (0)

* 'AI12-0367 Glitches in aspect specifications (0000-00-00)'

  The AI clarifies a few wording omissions. For example, a specified Small value for a fixed point type has to be positive.

  RM references: 3.05.09 (8/2) 3.05.10 (2/1) 13.01 (9.1/5) 13.14 (10)

* 'AI12-0368 Declare expressions can be static (2020-05-30)'

  AI12-0368 allows declare expressions to be static in Ada 2022.

  RM references: 4.09 (8) 4.09 (12.1/3) 4.09 (17) 6.01.01 (24.2/5) 6.01.01 (24.3/5) 6.01.01 (24.4/5) 6.01.01 (24.5/5) C.04 (9)

* 'AI12-0369 Relaxing barrier restrictions (2020-03-25)'

  The definitions of the `Simple_Barriers` and `Pure_Barriers` restrictions were modified by this AI, replacing uses of "statically denotes" with "statically names". This means that in many cases (but not all) a barrier expression that references a subcomponent of a component of the protected type while subject to either of the two restrictions is now allowed; with the previous restriction definitions, such a barrier expression would not have been legal.

  RM references: D.07 (1.3/5) D.07 (10.12/5)

* 'AI12-0372 Static accessibility of "master of the call" (2020-09-09)'

  Add an extra compile-time accessibility check for explicitly aliased parameters needed to prevent dangling references.

  RM references: 3.10.02 (10.5/5) 3.10.02 (19.3/4) 6.04.01 (6.4/3)

* 'AI12-0373 Bunch of fixes (0000-00-00)'

  Small clarifications to various RM entries with minor impact on compiler implementation.

  RM references: 3.01 (1) 4.02 (4) 4.02 (8/2) 4.02.01 (3/5) 4.02.01 (4/5) 4.02.01 (5/5) 4.09 (17.3/5) 6.01.01 (41/5) 8.05.04 (4/3) 13.01.01 (4/3) 13.01.01 (11/3) 13.14 (3/5)

* ‘AI12-0376 Representation changes finally allowed for untagged derived types (0000-00-00)’

  A change of representation for a derived type is allowed in some previously-illegal cases where a change of representation is required to implement a call to a derived subprogram.

  RM references: 13.01 (10/4)

* ‘AI12-0377 View conversions and out parameters of types with Default_Value revisited (2020-06-17)’

  This AI clarifies that an actual of an out parameter that is a view conversion is illegal if either the target or operand type has Default_Value specified while the other does not.

  RM references: 6.04.01 (5.1/4) 6.04.01 (5.2/4) 6.04.01 (5.3/4) 6.04.01 (13.1/4) 6.04.01 (13.2/4) 6.04.01 (13.3/4) 6.04.01 (13.4/4) 6.04.01 (15/3)

* ‘AI12-0381 Tag of a delta aggregate (0000-00-00)’

  In the case of a delta aggregate of a specific tagged type, the tag of the aggregate comes from the specific type (as opposed to somehow from the base object).

  RM references: 4.03.04 (14/5)

* ‘AI12-0382 Loosen type-invariant overriding requirement of AI12-0042-1 (0000-00-00)’

  The AI relaxes some corner-case legality rules about type invariants that were added by AI12-0042-1.

  RM references: 7.3.2(6.1/4)

* ‘AI12-0383 Renaming values (2020-06-17)’

  This AI allow names that denote values rather than objects to nevertheless be renamed using an object renaming.

  RM references: 8.05.01 (1) 8.05.01 (4) 8.05.01 (4.1/2) 8.05.01 (6/2) 8.05.01 (8)

* ‘AI12-0384-2 Fixups for Put_Image and Text_Buffers (2021-04-29)’

  In GNAT’s initial implementation of the Ada 2022 `Put_Image` aspect and attribute, buffering was performed using a GNAT-defined package, `Ada.Strings.Text_Output`. Ada 2022 requires a different package, Ada.“Strings.Text_Buffers“, for this role, and that package is now provided, and the older package is eliminated.

  RM references: 4.10 (0) A.04.12 (0)

* ‘AI12-0385 Predefined shifts and rotates should be static (0000-00-00)’

  This AI allows Shift and Rotate operations in static expressions. GNAT implements this AI partially.

  RM references: 4.09 (20)

* ‘AI12-0389 Ignoring unrecognized aspects (2020-10-08)’

  Two new restrictions, `No_Unrecognized_Aspects` and `No_Unrecognized_Pragmas`, are available to make the compiler emit error messages on unrecognized pragmas and aspects.

  RM references: 13.01.01 (38/3) 13.12.01 (6.3/3)

* 'AI12-0394 Named Numbers and User-Defined Numeric Literals (2020-10-05)'

  Ada 2022 allows using integer named numbers with types that have an `Integer_Literal` aspect. Similarly, real named numbers may now be used with types that have a `Real_Literal` aspect with an overloading that takes two strings, to be used in particular with `Ada.Numerics.Big_Numbers.Big_Reals`.

  RM references: 3.03.02 (3) 4.02.01 (4/5) 4.02.01 (8/5) 4.02.01 (12/5) 4.02.01 (13/5) 4.09 (5)

* 'AI12-0395 Allow aspect_specifications on formal parameters (0000-00-00)'

  Change syntax rules to allow aspect_specifications on formal parameters, if an implementation if an implementation wants to define one. Currently, GNAT doesn't define any such aspect_specifications.

  RM references: 6.01 (15/3)

* 'AI12-0397 Default_Initial_Condition applied to derived type (2020-12-09)'

  The compiler now implements the rules for resolving `Default_Initial_Condition` expressions that involve references to the current instance of types with the aspect, as specified by this AI. The type of the current instance is defined to be like a formal derived type, so for a derived type that inherits the aspect, a call passing the current instance to a primitive means that the call will resolve to invoke the corresponding primitive of the descendant type. This also now permits calls to abstract primitives to occur within the aspect expression of an abstract type.

  RM references: 7.03.03 (3/5) 7.03.03 (6/5) 7.03.03 (8/5)

* 'AI12-0398 Most declarations should have aspect specifications (2020-11-19)'

  It is now possible to specify aspects for discriminant specifications, extended return object declarations, and entry index specifications. This is an extension added for Ada 2022 by this AI.

  RM references: 3.07 (5/2) 6.03.01 (25) 6.05 (2.1/3) 9.05.02 (8)

* 'AI12-0399 Aspect specification for Preelaborable_Initialization (0000-00-00)'

  Semantics-preserving presentation change. Replace `Preelaborable_Initialization` pragmas with equivalent aspect specs in the listed predefined packages. GNAT follows the guidance of this AI partially.

  RM references: 9.05 (53/5) 3.09 (6/5) 7.06 (5/2) 7.06 (7/2) 11.04.01 (2/5) 11.04.01 (3/2) 13.11 (6/2) 13.11.04 (4/5) 13.11.04 (5/5) 13.13.01 (3/2) A.04.02 (4/2) A.04.02 (20/2) A.04.05 (4/2) A.04.07 (4/2) A.04.07 (20/2) A.04.08 (4/2) A.04.08 (20/2) A.12.01 (5/4) A.18.02 (8/5) A.18.02 (9/2) A.18.02 (79.2/5) A.18.02 (79.3/5) A.18.03 (6/5) A.18.03 (7/2) A.18.03 (50.2/5) A.18.03 (50.3/5) A.18.05 (3/5) A.18.05 (4/2) A.18.05 (37.3/5) A.18.05 (37.4/5) A.18.06 (4/5) A.18.06 (5/2) A.18.06 (51.4/5) A.18.06 (51.5/5) A.18.08 (3/5) A.18.08 (4/2) A.18.08 (58.2/5) A.18.08 (58.3/5) A.18.09 (4/5) A.18.09 (5/2) A.18.09 (74.2/5) A.18.09 (74.3/5) A.18.10 (8/5) A.18.10 (9/3) A.18.10 (70.2/5) A.18.10 (70.3/5) A.18.18 (6/5) B.03.01 (5/2) C.07.01 (2/5) G.01.01 (4/2)

* 'AI12-0400 Ambiguities associated with Vector Append and container aggregates (0000-00-00)'

  Change the names of subprograms in the predefined Vector containers from `Append` to `Append_Vector` and from `Prepend` to `Prepend_Vector` in order to resolve some ambiguity problems. GNAT adds the subprograms with new names but also keeps the old ones for backward compatibility.

RM references: A.18.02 (8/5) A.18.02 (36/5) A.18.02 (37/5) A.18.02 (38/5) A.18.02 (44/5) A.18.02 (46/5) A.18.02 (47/5) A.18.02 (58/5) A.18.02 (79.2/5) A.18.02 (150/5) A.18.02 (151/5) A.18.02 (152/5) A.18.02 (153/5) A.18.02 (154/5) A.18.02 (155/5) A.18.02 (156/5) A.18.02 (168/5) A.18.02 (169/5) A.18.02 (172/5) A.18.02 (173/5) A.18.02 (174/5) A.18.02 (175.1/5) A.18.03 (23/5) A.18.03 (23.1/5) A.18.03 (58.2/5) A.18.03 (96/5) A.18.03 (97.1/5)

* 'AI12-0401 Renaming of qualified expression of variable (2020-10-31)'

  Ada 2022 AI12-0401 restricts renaming of a qualified expression to cases where the operand is a constant, or the target subtype statically matches the nominal subtype of the operand, or is unconstrained with no predicates, to prevent setting variables to values outside their range or constraints.

  RM references: 3.03 (23.2/3) 8.05.01 (4.7/5) 8.05.01 (5/3)

* 'AI12-0402 Master of a function call with elementary result type (2026-01-07)'

  This AI restricts the rule that makes the master of a function call, which is used to directly initialize part of an object, that of the object being initialized, to the case where the result type of the function is composite.

  RM references: 3.10.2 (10.2/5)

* 'AI12-0409 Preelaborable_Initialization and bounded containers (2021-06-23)'

  As defined by this AI, the `Preelaborable_Initializatio` aspect now has a corresponding attribute of the same name. Types declared within a generic package specification are permitted to specify the expression of a `Prelaborable_Initialization` aspect by including one or more references to the attribute applied to a formal private or formal derived type conjoined by `and` operators. This permits the full type of a private type with such an aspect expression to have components of the named formal types, and such a type will have preelaborable initialization in an instance when the actual types for all referenced formal types have preelaborable initialization.

  RM references: 10.02.01 (4.1/2) 10.02.01 (4.2/2) 10.02.01 (11.1/2) 10.02.01 (11.2/2) 10.02.01 (11.6/2) 10.02.01 (11.7/2) 10.02.01 (11.8/2) 13.01 (11/3) A.18.19 (5/5) A.18.20 (5/5) A.18.21 (5/5) A.18.22 (5/5) A.18.23 (5/5) A.18.24 (5/5) A.18.25 (5/5) A.18.32 (6/5) J.15.14 (0)

* 'AI12-0411 Add "bool" to Interfaces.C (0000-00-00)'

  RM references: B.03 (13) B.03 (43/2) B.03 (65.1/4)

* 'AI12-0412 Abstract Pre/Post'Class on primitive of abstract type (2021-05-19)'

  In Ada 2022, by AI12-0412, it's legal to specify Pre'Class and Post'Class aspects on nonabstract primitive subprograms of an abstract type, but if the expression of such an aspect is nonstatic, then it's illegal to make a nondispatching call to such a primitive, to apply `'Access` to it, or to pass such a primitive as an actual subprogram for a concrete formal subprogram in a generic instantiation.

  RM references: 6.01.01 (18.2/4)

* 'AI12-0413 Reemergence of "=" when defined to be abstract (0000-00-00)'

  The AI clarifies rules about operator reemergence in instances, and nondispatching calls to abstract subprograms.

RM references: 3.09.03 (7) 4.05.02 (14.1/3) 4.05.02 (24.1/3) 12.05 (8/3)

* 'AI12-0423 Aspect inheritance fixups (0000-00-00)'

  Clarify that the No_Return aspect behaves as one would expect for an inherited subprogram and that inheritance works as one would expect for a multi-part aspect whose value is specified via an aggregate (e.g., the Aggregate aspect).

  RM references: 6.05.01 (3.3/3) 13.01 (15.7/5) 13.01 (15.8/5)

* 'AI12-0432 View conversions of assignments and predicate checks (2021-05-05)'

  When a predicate applies to a tagged type, a view conversion to that type normally requires a predicate check. However, as specified by AI12-0432, when the view conversion appears as the target of an assignment, a predicate check is not applied to the object in the conversion.

  RM references: 3.02.04 (31/5) 4.06 (51.1/5)

17 GNAT language extensions

The GNAT compiler implements a certain number of language extensions on top of the latest Ada standard, implementing its own extended superset of Ada.

There are two sets of language extensions:

* The first is the curated set. The features in that set are features that we consider being worthy additions to the Ada language, and that we want to make available to users early on.
* The second is the experimental set. It includes the first, but also experimental features, which are considered experimental because they're still in an early prototyping phase. These features might be removed or heavily modified at any time.

17.1 How to activate the extended GNAT Ada superset

There are two ways to activate the extended GNAT Ada superset:

* The [Pragma Extensions_Allowed], page 35. To activate the curated set of extensions, you should use

```
pragma Extensions_Allowed (On)
```

As a configuration pragma, you can either put it at the beginning of a source file, or in a `.adc` file corresponding to your project.

* The `-gnatX` command-line option will activate the curated subset of extensions.

> **Attention:** You can activate the experimental set of extensions in addition by using either the `-gnatX0` command-line option, or the pragma `Extensions_Allowed` with `All_Extensions` as an argument. However, it is not recommended you use this subset for serious projects; it is only meant as a technology preview for use in playground experiments.

17.2 Curated Extensions

Features activated via `-gnatX` or `pragma Extensions_Allowed (On)`.

17.2.1 Local Declarations Without Block

A `basic_declarative_item` may appear at the place of any statement. This avoids the heavy syntax of block_statements just to declare something locally.

The only valid kinds of declarations for now are `object_declaration`, `object_renaming_declaration`, `use_package_clause`, and `use_type_clause`.

For example:

```
if X > 5 then
   X := X + 1;

   Squared : constant Integer := X**2;
```

```
    X := X + Squared;
end if;
```

It is generally a good practice to declare local variables (or constants) with as short a lifetime as possible. However, introducing a declare block to accomplish this is a relatively heavy syntactic load along with a traditional extra level of indentation. The alternative syntax supported here allows declarations in any statement sequence. The lifetime of such local declarations is until the end of the enclosing construct. The same enclosing construct cannot contain several declarations of the same defining name; however, overriding symbols from higher-level scopes works similarly to the explicit `declare` block.

If the enclosing construct allows an exception handler (such as an accept statement, begin-except-end block or a subprogram body), declarations that appear at the place of a statement are 'not' visible within the handler. Only declarations that precede the beginning of the construct with an exception handler would be visible in this handler.

Attention: Here are a couple of examples illustrating the scoping rules described above.

1. Those declarations are not visible from the potential exception handler:

   ```
   begin
      A : Integer
      ...
   exception
      when others =>
          Put_Line (A'Image) --  ILLEGAL
   end;
   ```

2. The following is legal

   ```
   declare
      A : Integer := 10;
   begin
      A : Integer := 12;
   end;
   ```

   because it is roughly expanded into

   ```
   declare
      A : Integer := 10;
   begin
      declare
         A : Integer := 12;
      begin
         ...
      end;
   end;

   And as such the second ``A`` declaration is hiding the first one.
   ```

17.2.2 Deep delta Aggregates

Ada 2022's delta aggregates are extended to allow deep updates.

A delta aggregate may be used to specify new values for subcomponents of the copied base value, instead of only new values for direct components of the copied base value. This allows a more compact expression of updated values with a single delta aggregate, instead of multiple nested delta aggregates.

The syntax of delta aggregates in the extended version is the following:

17.2.2.1 Syntax

```
delta_aggregate ::= record_delta_aggregate | array_delta_aggregate

record_delta_aggregate ::=
  ( base_expression with delta record_subcomponent_association_list )

record_subcomponent_association_list ::=
  record_subcomponent_association {, record_subcomponent_association}

record_subcomponent_association ::=
  record_subcomponent_choice_list => expression

record_subcomponent_choice_list ::=
  record_subcomponent_choice {'|' record_subcomponent_choice}

record_subcomponent_choice ::=
    component_selector_name
  | record_subcomponent_choice (expression)
  | record_subcomponent_choice . component_selector_name

array_delta_aggregate ::=
    ( base_expression with delta array_component_association_list )
  | '[' base_expression with delta array_component_association_list ']'
  | ( base_expression with delta array_subcomponent_association_list )
  | '[' base_expression with delta array_subcomponent_association_list ']'

array_subcomponent_association_list ::=
  array_subcomponent_association {, array_subcomponent_association}

array_subcomponent_association ::=
  array_subcomponent_choice_list => expression

array_subcomponent_choice_list ::=
  array_subcomponent_choice {'|' array_subcomponent_choice}

array_subcomponent_choice ::=
    ( expression )
  | array_subcomponent_choice (expression)
```

```
| array_subcomponent_choice . component_selector_name
```

17.2.2.2 Legality Rules

1. For an `array_delta_aggregate`, the discrete_choice shall not be 'others'.
2. For an `array_delta_aggregate`, the dimensionality of the type of the `delta_aggregate` shall be 1.
3. For an `array_delta_aggregate`, the `base_expression` and each expression in every `array_component_association` or `array_subcomponent_association` shall be of a nonlimited type.
4. For a `record_delta_aggregate`, no `record_subcomponent_choices` that consists of only `component_selector_names` shall be the same or a prefix of another record_subcomponent_choice.
5. For an `array_subcomponent_choice` or a `record_subcomponent_choice` the `component_selector_name` shall not be a subcomponent that depends on discriminants of an unconstrained record subtype with defaulted discriminants unless its prefix consists of only `component_selector_names`.

   [Rationale: As a result of this rule, accessing the subcomponent can only lead to a discriminant check failure if the subcomponent was not present in the object denoted by the base_expression, prior to any update.]

17.2.2.3 Dynamic Semantics

The evaluation of a `delta_aggregate` begins with the evaluation of the `base_expression` of the delta_aggregate; then that value is used to create and initialize the anonymous object of the aggregate. The bounds of the anonymous object of an `array_delta_aggregate` and the discriminants (if any) of the anonymous object of a `record_delta_aggregate` are those of the `base_expression`.

If a `record_delta_aggregate` is of a specific tagged type, its tag is that of the specific type; if it is of a class-wide type, its tag is that of the base_expression.

For a `delta_aggregate`, for each `discrete_choice` or each subcomponent associated with a `record_subcomponent_associated`, `array_component_association` or `array_subcomponent_association` (in the order given in the enclosing `discrete_choice_list` or `subcomponent_association_list`, respectively):

- if the associated subcomponent belongs to a variant, a check is made that the values of the governing discriminants are such that the anonymous object has this component. The exception `Constraint_Error` is raised if this check fails.
- if the associated subcomponent is a subcomponent of an array, then for each represented index value (in ascending order, if the `discrete_choice` represents a range):
  * the index value is converted to the index type of the array type.
  * a check is made that the index value belongs to the index range of the corresponding array part of the anonymous object; `Constraint_Error` is raised if this check fails.
  * the expression of the `record_subcomponent_association`, `array_component_association` or `array_subcomponent_association` is evaluated, converted to the nominal subtype

of the associated subcomponent, and assigned to the corresponding subcomponent of the anonymous object.

17.2.2.4 Examples

```
declare
   type Point is record
      X, Y : Integer;
   end record;

   type Segment is array (1 .. 2) of Point;
   type Triangle is array (1 .. 3) of Segment;

   S : Segment := (1 .. 2 => (0, 0));
   T : Triangle := (1 .. 3 => (1 .. 2 => (0, 0)));
begin
   S := (S with delta (1).X | (2).Y => 12, (1).Y => 15);

   pragma Assert (S (1).X = 12);
   pragma Assert (S (2).Y = 12);
   pragma Assert (S (1).Y = 15);

   T := (T with delta (2)(1).Y => 18);
   pragma Assert (T (2)(1).Y = 18);
end;
```

17.2.3 Fixed lower bounds for array types and subtypes

Unconstrained array types and subtypes can be specified with a lower bound that is fixed to a certain value, by writing an index range that uses the syntax `<lower-bound-expression> .. <>`. This guarantees that all objects of the type or subtype will have the specified lower bound.

For example, a matrix type with fixed lower bounds of zero for each dimension can be declared by the following:

```
type Matrix is
  array (Natural range 0 .. <>, Natural range 0 .. <>) of Integer;
```

Objects of type `Matrix` declared with an index constraint must have index ranges starting at zero:

```
M1 : Matrix (0 .. 9, 0 .. 19);
M2 : Matrix (2 .. 11, 3 .. 22);  -- Warning about bounds; will raise CE
```

Similarly, a subtype of `String` can be declared that specifies the lower bound of objects of that subtype to be `1`:

```
subtype String_1 is String (1 .. <>);
```

If a string slice is passed to a formal of subtype `String_1` in a call to a subprogram `S`, the slice's bounds will "slide" so that the lower bound is `1`.

Within `S`, the lower bound of the formal is known to be `1`, so, unlike a normal unconstrained `String` formal, there is no need to worry about accounting for other possible lower-bound values. Sliding of bounds also occurs in other contexts, such as for object declarations with an unconstrained subtype with fixed lower bound, as well as in subtype conversions.

Use of this feature increases safety by simplifying code, and can also improve the efficiency of indexing operations, since the compiler statically knows the lower bound of unconstrained array formals when the formal's subtype has index ranges with static fixed lower bounds.

17.2.4 Prefixed-view notation for calls to primitive subprograms of untagged types

When operating on an untagged type, if it has any primitive operations, and the first parameter of an operation is of the type (or is an access parameter with an anonymous type that designates the type), you may invoke these operations using an `object.op(...)` notation, where the parameter that would normally be the first parameter is brought out front, and the remaining parameters (if any) appear within parentheses after the name of the primitive operation.

This same notation is already available for tagged types. This extension allows for untagged types. It is allowed for all primitive operations of the type independent of whether they were originally declared in a package spec, or were inherited and/or overridden as part of a derived type declaration occurring anywhere, so long as the first parameter is of the type, or an access parameter designating the type.

For example:

```
generic
   type Elem_Type is private;
package Vectors is
    type Vector is private;
    procedure Add_Element (V : in out Vector; Elem : Elem_Type);
    function Nth_Element (V : Vector; N : Positive) return Elem_Type;
    function Length (V : Vector) return Natural;
private
    function Capacity (V : Vector) return Natural;
       --  Return number of elements that may be added without causing
       --  any new allocation of space

    type Vector is ...
      with Type_Invariant => Vector.Length <= Vector.Capacity;
    ...
end Vectors;

package Int_Vecs is new Vectors(Integer);

V : Int_Vecs.Vector;
...
V.Add_Element(42);
V.Add_Element(-33);

pragma Assert (V.Length = 2);
pragma Assert (V.Nth_Element(1) = 42);
```

17.2.5 Expression defaults for generic formal functions

The declaration of a generic formal function is allowed to specify an expression as a default, using the syntax of an expression function.

Here is an example of this feature:

```
generic
   type T is private;
   with function Copy (Item : T) return T is (Item); -- Defaults to Item
package Stacks is

   type Stack is limited private;

   procedure Push (S : in out Stack; X : T); -- Calls Copy on X
   function Pop (S : in out Stack) return T; -- Calls Copy to return item

private
   -- ...
end Stacks;
```

If Stacks is instantiated with an explicit actual for Copy, then that will be called when Copy is called in the generic body. If the default is used (i.e. there is no actual corresponding to Copy), then calls to Copy in the instance will simply return Item.

17.2.6 String interpolation

The syntax for string literals is extended to support string interpolation. An interpolated string literal starts with `f`, immediately before the first double-quote character.

Within an interpolated string literal, an arbitrary expression, when enclosed in `{ ... }`, is expanded at run time into the result of calling `'Image` on the result of evaluating the expression enclosed by the brace characters, unless it is already a string or a single character.

Here is an example of this feature where the expressions `Name` and `X + Y` will be evaluated and included in the string.

```
procedure Test_Interpolation is
   X    : Integer := 12;
   Y    : Integer := 15;
   Name : String := "Leo";
begin
   Put_Line (f"The name is {Name} and the sum is {X + Y}.");
end Test_Interpolation;
```

This will print:

```
The name is Leo and the sum is 27.
```

In addition, an escape character (\) is provided for inserting certain standard control characters (such as `\t` for tabulation or `\n` for newline) or to escape characters with special significance to the interpolated string syntax, namely `"`, `{`, `}`,and `\` itself.

escaped_character	meaning

`\a`	ALERT
`\b`	BACKSPACE
`\f`	FORM FEED
`\n`	LINE FEED
`\r`	CARRIAGE RETURN
`\t`	CHARACTER TABULATION
`\v`	LINE TABULATION
`\0`	NUL
`\\`	`\`
`\"`	`"`
`\{`	`{`
`\}`	`}`

Note that, unlike normal string literals, doubled double-quote characters have no special significance. So to include a double-quote or a brace character in an interpolated string, they must be preceded by a `\`. Multiple interpolated strings are concatenated. For example:

```
Put_Line
  (f"X = {X} and Y = {Y} and X+Y = {X+Y};\n" &
   f" a double quote is \" and" &
   f" an open brace is \{");
```

This will print:

```
X = 12 and Y = 15 and X+Y = 27
a double quote is " and an open brace is {
```

17.2.7 Constrained attribute for generic objects

The `Constrained` attribute is permitted for objects of generic types. The result indicates whether the corresponding actual is constrained.

17.2.8 `Static` aspect on intrinsic functions

The Ada 202x `Static` aspect can be specified on Intrinsic imported functions and the compiler will evaluate some of these intrinsics statically, in particular the `Shift_Left` and `Shift_Right` intrinsics.

17.2.9 First Controlling Parameter

A new pragma/aspect, `First_Controlling_Parameter`, is introduced for tagged types, altering the semantics of primitive/controlling parameters. When a tagged type is marked with this aspect, only subprograms where the first parameter is of that type will be considered dispatching primitives. This pragma/aspect applies to the entire hierarchy, starting from the specified type, without affecting inherited primitives.

Here is an example of this feature:

```
package Example is
   type Root is tagged private;

   procedure P (V : Integer; V2 : Root);
   -- Primitive

   type Child is tagged private
     with First_Controlling_Parameter;

private
   type Root is tagged null record;
   type Child is new Root with null record;

   overriding
   procedure P (V : Integer; V2 : Child);
   -- Primitive

   procedure P2 (V : Integer; V2 : Child);
   -- NOT Primitive

   function F return Child; -- NOT Primitive

   function F2 (V : Child) return Child;
   -- Primitive, but only controlling on the first parameter
end Example;
```

Note that `function F2 (V : Child) return Child;` differs from `F2 (V : Child) return Child'Class;` in that the return type is a specific, definite type. This is also distinct from the legacy semantics, where further derivations with added fields would require overriding the function.

The option `-gnatw_j`, that you can pass to the compiler directly, enables warnings related to this new language feature. For instance, compiling the example above without this switch produces no warnings, but compiling it with `-gnatw_j` generates the following warning on the declaration of procedure P2:

```
warning: not a dispatching primitive of tagged type "Child"
warning: disallowed by First_Controlling_Parameter on "Child"
```

For generic formal tagged types, you can specify whether the type has the First_Controlling_Parameter aspect enabled:

```
generic
```

```
   type T is tagged private with First_Controlling_Parameter;
package T is
    type U is new T with null record;
    function Foo return U; -- Not a primitive
end T;
```

For tagged partial views, the value of the aspect must be consistent between the partial and full views:

```
package R is
   type T is tagged private;
...
private
   type T is tagged null record with First_Controlling_Parameter; -- ILLEGAL
end R;
```

Restricting the position of controlling parameter offers several advantages:

* Simplification of the dispatching rules improves readability of Ada programs. One doesn't need to analyze all subprogram parameters to understand if the given subprogram is a primitive of a certain tagged type.
* A programmer is free to use any type, including class-wide types, on other parameters of a subprogram, without the need to consider possible effects of overriding a primitive or creating new one.
* The result of a function is never a controlling result.

17.2.10 Generalized Finalization

The `Finalizable` aspect can be applied to any record type, tagged or not, to specify that it provides the same level of control on the operations of initialization, finalization, and assignment of objects as the controlled types (see RM 7.6(2) for a high-level overview). The only restriction is that the record type must be a root type, in other words not a derived type.

The aspect additionally makes it possible to specify relaxed semantics for the finalization operations by means of the `Relaxed_Finalization` setting. Here is the archetypal example:

```
type T is record
  ...
end record
  with Finalizable => (Initialize           => Initialize,
                       Adjust               => Adjust,
                       Finalize             => Finalize,
                       Relaxed_Finalization => True);

procedure Adjust     (Obj : in out T);
procedure Finalize   (Obj : in out T);
procedure Initialize (Obj : in out T);
```

The three procedures have the same profile, with a single `in out` parameter, and also have the same dynamic semantics as for controlled types:

- `Initialize` is called when an object of type `T` is declared without initialization expression.

- Adjust is called after an object of type T is assigned a new value.
- Finalize is called when an object of type T goes out of scope (for stack-allocated objects) or is deallocated (for heap-allocated objects). It is also called when the value is replaced by an assignment.

However, when Relaxed_Finalization is either True or not explicitly specified, the following differences are implemented relative to the semantics of controlled types:

* The compiler has permission to perform no automatic finalization of heap-allocated objects: Finalize is only called when such an object is explicitly deallocated, or when the designated object is assigned a new value. As a consequence, no runtime support is needed for performing implicit deallocation. In particular, no per-object header data is needed for heap-allocated objects.

  Heap-allocated objects allocated through a nested access type will therefore 'not' be deallocated either. The result is simply that memory will be leaked in this case.
* The Adjust and Finalize procedures are automatically considered as having the [No_Raise aspect], page 356, specified for them. In particular, the compiler has permission to enforce none of the guarantees specified by the RM 7.6.1 (14/1) and subsequent subclauses.

Simple example of ref-counted type:

```
type T is record
   Value     : Integer;
   Ref_Count : Natural := 0;
end record;

procedure Inc_Ref (X : in out T);
procedure Dec_Ref (X : in out T);

type T_Access is access all T;

type T_Ref is record
   Value : T_Access;
end record
   with Finalizable => (Adjust   => Adjust,
                        Finalize => Finalize);

procedure Adjust (Ref : in out T_Ref) is
begin
   Inc_Ref (Ref.Value);
end Adjust;

procedure Finalize (Ref : in out T_Ref) is
begin
   Def_Ref (Ref.Value);
end Finalize;
```

Simple file handle that ensures resources are properly released:

```
package P is
```

```
   type File (<>) is limited private;

   function Open (Path : String) return File;

   procedure Close (F : in out File);

private
   type File is limited record
      Handle : ...;
   end record
      with Finalizable (Finalize => Close);
end P;
```

17.2.10.1 Finalizable tagged types

The aspect is inherited by derived types and the primitives may be overridden by the derivation. The compiler-generated calls to these operations are then dispatching whenever it makes sense, i.e. when the object in question is of a class-wide type and the class includes at least one finalizable tagged type.

17.2.10.2 Composite types

When a finalizable type is used as a component of a composite type, the latter becomes finalizable as well. The three primitives are derived automatically in order to call the primitives of their components. The dynamic semantics is the same as for controlled components of composite types.

17.2.10.3 Interoperability with controlled types

Finalizable types are fully interoperable with controlled types, in particular it is possible for a finalizable type to have a controlled component and vice versa, but the stricter dynamic semantics, in other words that of controlled types, is applied in this case.

17.3 Experimental Language Extensions

Features activated via `-gnatX0` or `pragma Extensions_Allowed (All_Extensions)`.

17.3.1 Conditional when constructs

This feature extends the use of `when` as a way to condition a control-flow related statement, to all control-flow related statements.

To do a conditional return in a procedure the following syntax should be used:

```
procedure P (Condition : Boolean) is
begin
   return when Condition;
end P;
```

This will return from the procedure if `Condition` is true.

When being used in a function the conditional part comes after the return value:

```
function Is_Null (I : Integer) return Boolean is
begin
```

```
   return True when I = 0;
   return False;
end;
```

In a similar way to the `exit when` a `goto ... when` can be employed:

```
procedure Low_Level_Optimized is
   Flags : Bitmapping;
begin
   Do_1 (Flags);
   goto Cleanup when Flags (1);

   Do_2 (Flags);
   goto Cleanup when Flags (32);

   --  ...

<<Cleanup>>
   --  ...
end;
```

To use a conditional raise construct:

```
procedure Foo is
begin
   raise Error when Imported_C_Func /= 0;
end;
```

An exception message can also be added:

```
procedure Foo is
begin
   raise Error with "Unix Error"
     when Imported_C_Func /= 0;
end;
```

17.3.2 Implicit With

This feature allows a standalone `use` clause in the context clause of a compilation unit to imply an implicit `with` of the same library unit where an equivalent `with` clause would be allowed.

```
use Ada.Text_IO;
procedure Main is
begin
   Put_Line ("Hello");
end;
```

17.3.3 Storage Model

This extends Storage Pools into a more efficient model allowing higher performance, easier integration with low footprint embedded run-times and copying data between different pools of memory. The latter is especially useful when working with distributed memory models, in particular to support interactions with GPU.

17.3.3.1 Aspect Storage_Model_Type

A Storage model is a type with a specified `Storage_Model_Type` aspect, e.g.:

```
type A_Model is null record
   with Storage_Model_Type (...);
```

Storage_Model_Type itself accepts six parameters:

- Address_Type, the type of the address managed by this model. This has to be a scalar type or derived from System.Address.
- Allocate, a procedure used for allocating memory in this model
- Deallocate, a procedure used for deallocating memory in this model
- Copy_To, a procedure used to copy memory from native memory to this model
- Copy_From, a procedure used to copy memory from this model to native memory
- Storage_Size, a function returning the amount of memory left
- Null_Address, a value for the null address value

By default, Address_Type is System.Address, and the five subprograms perform native operations (e.g. the allocator is the native `new` allocator). Users can decide to specify one or more of these. When an Address_Type is specified to be other than System.Address, all of the subprograms have to be specified.

The prototypes of these procedures are as follows:

```
procedure Allocate
  (Model           : in out A_Model;
   Storage_Address : out Address_Type;
   Size            : Storage_Count;
   Alignment       : Storage_Count);

procedure Deallocate
  (Model           : in out A_Model;
   Storage_Address : out Address_Type;
   Size            : Storage_Count;
   Alignment       : Storage_Count);

procedure Copy_To
  (Model   : in out A_Model;
   Target  : Address_Type;
   Source  : System.Address;
   Size    : Storage_Count);

procedure Copy_From
  (Model  : in out A_Model;
   Target : System.Address;
   Source : Address_Type;
   Size   : Storage_Count);

function Storage_Size
  (Pool : A_Model)
```

```
   return Storage_Count;
```

Here's an example of how this could be instantiated in the context of CUDA:

```
package CUDA_Memory is

   type CUDA_Storage_Model is null record
      with Storage_Model_Type => (
         Address_Type => CUDA_Address,
         Allocate     => CUDA_Allocate,
         Deallocate   => CUDA_Deallocate,
         Copy_To      => CUDA_Copy_To,
         Copy_From    => CUDA_Copy_From,
         Storage_Size => CUDA_Storage_Size,
         Null_Address => CUDA_Null_Address
      );

   type CUDA_Address is new System.Address;
   --  We're assuming for now same address size on host and device

   procedure CUDA_Allocate
     (Model           : in out CUDA_Storage_Model;
      Storage_Address : out CUDA_Address;
      Size            : Storage_Count;
      Alignment       : Storage_Count);

   procedure CUDA_Deallocate
     (Model           : in out CUDA_Storage_Model;
      Storage_Address : out CUDA_Address;
      Size            : Storage_Count;
      Alignment       : Storage_Count);

   procedure CUDA_Copy_To
     (Model  : in out CUDA_Storage_Model;
      Target : CUDA_Address;
      Source : System.Address;
      Size   : Storage_Count);

   procedure CUDA_Copy_From
     (Model   : in out CUDA_Storage_Model;
      Target  : System.Address;
      Source  : CUDA_Address;
      Size    : Storage_Count);

   function CUDA_Storage_Size
     (Pool : CUDA_Storage_Model)
      return Storage_Count return Storage_Count'Last;
```

```
   CUDA_Null_Address : constant CUDA_Address :=
      CUDA_Address (System.Null_Address);

   CUDA_Memory : CUDA_Storage_Model;

end CUDA_Memory;
```

17.3.3.2 Aspect Designated_Storage_Model

A new aspect, Designated_Storage_Model, allows to specify the memory model for the objects pointed by an access type. Under this aspect, allocations and deallocations will come from the specified memory model instead of the standard ones. In addition, if write operations are needed for initialization, or if there is a copy of the target object from and to a standard memory area, the Copy_To and Copy_From functions will be called. It allows to encompass the capabilities of storage pools, e.g.:

```
procedure Main is
   type Integer_Array is array (Integer range <>) of Integer;

   type Host_Array_Access is access all Integer_Array;
   type Device_Array_Access is access Integer_Array
      with Designated_Storage_Model => CUDA_Memory;

   procedure Free is new Unchecked_Deallocation
      (Host_Array_Type, Host_Array_Access);
   procedure Free is new Unchecked_Deallocation
      (Device_Array_Type, Device_Array_Access);

   Host_Array : Host_Array_Access := new Integer_Array (1 .. 10);

   Device_Array : Device_Array_Access := new Host_Array (1 .. 10);
   --  Calls CUDA_Storage_Model.Allocate to allocate the fat pointers and
   --  the bounds, then CUDA_Storage_Model.Copy_In to copy the values of the
   --  boundaries.
begin
   Host_Array.all := (others => 0);

   Device_Array.all := Host_Array.all;
   --  Calls CUDA_Storage_Model.Copy_To to write to the device array from the
   --  native memory.

   Host_Array.all := Device_Array.all;
   --  Calls CUDA_Storage_Model.Copy_From to read from the device array and
   --  write to native memory.

   Free (Host_Array);

   Free (Device_Array);
```

```
     --  Calls CUDA_Storage_Model.Deallocate;
   end;
```

Taking `'Address` of an object with a specific memory model returns an object of the type of the address for that memory category, which may be different from System.Address.

When copying is performed between two specific memory models, the native memory is used as a temporary between the two. E.g.:

```
type Foo_I is access Integer with Designated_Storage_Model => Foo;
type Bar_I is access Integer with Designated_Storage_Model => Bar;

  X : Foo_I := new Integer;
  Y : Bar_I := new Integer;
begin
  X.all := Y.all;
```

conceptually becomes:

```
  X : Foo_I := new Integer;
  T : Integer;
  Y : Bar_I := new Integer;
begin
  T := Y.all;
  X.all := T;
```

17.3.3.3 Legacy Storage Pools

Legacy Storage Pools are now replaced by a Storage_Model_Type. They are implemented as follows:

```
type Root_Storage_Pool is abstract
  new Ada.Finalization.Limited_Controlled with private
with Storage_Model_Type => (
   Allocate     => Allocate,
   Deallocate   => Deallocate,
   Storage_Size => Storage_Size
);
pragma Preelaborable_Initialization (Root_Storage_Pool);

procedure Allocate
  (Pool                     : in out Root_Storage_Pool;
   Storage_Address          : out System.Address;
   Size_In_Storage_Elements : System.Storage_Elements.Storage_Count;
   Alignment                : System.Storage_Elements.Storage_Count)
is abstract;

procedure Deallocate
  (Pool                     : in out Root_Storage_Pool;
   Storage_Address          : System.Address;
   Size_In_Storage_Elements : System.Storage_Elements.Storage_Count;
   Alignment                : System.Storage_Elements.Storage_Count)
```

```
is abstract;

function Storage_Size
  (Pool : Root_Storage_Pool)
   return System.Storage_Elements.Storage_Count
is abstract;
```

The legacy notation:

```
type My_Pools is new Root_Storage_Pool with record [...]

My_Pool_Instance : Storage_Model_Pool.Storage_Model :=
   My_Pools'(others => <>);

type Acc is access Integer_Array with Storage_Pool => My_Pool;
```

can still be accepted as a shortcut for the new syntax.

17.3.4 Attribute Super

The `Super` attribute can be applied to objects of tagged types in order to obtain a view conversion to the most immediate specific parent type.

It cannot be applied to objects of types without any ancestors.

```
type T1 is tagged null record;
procedure P (V : T1);

type T2 is new T1 with null record;

type T3 is new T2 with null record;
procedure P (V : T3);

procedure Call (
  V1 : T1'Class;
  V2 : T2'Class;
  V3 : T3'Class) is
begin
  V1'Super.P; --  Illegal call as T1 doesn't have any ancestors
  V2'Super.P; --  Equivalent to "T1 (V).P;", a non-dispatching call
              --  to T1's primitive procedure P.
  V3'Super.P; --  Equivalent to "T2 (V).P;"; Since T2 doesn't
              --  override P, a non-dispatching call to T1.P is
              --  executed.
end;
```

17.3.5 Simpler Accessibility Model

The goal of this feature is to simplify the accessibility rules by removing dynamic accessibility checks that are often difficult to understand and debug. The new rules eliminate the need for runtime accessibility checks by imposing more conservative legality rules when enabled via a new restriction (see RM 13.12), No_Dynamic_Accessibility_Checks, which prevents dangling reference problems at compile time.

This restriction has no effect on the user-visible behavior of a program when executed; the only effect of this restriction is to enable additional compile-time checks (described below) which ensure statically that Ada's dynamic accessibility checks will not fail.

The feature can be activated with `pragma Restrictions (No_Dynamic_Accessibility_Checks);`. As a result, additional compile-time checks are performed; these checks pertain to stand-alone objects, subprogram parameters, and function results as described below.

All of the refined rules are compatible with the [use of anonymous access types in SPARK] (‘`http://docs.adacore.com/spark2014-docs/html/lrm/declarations-and-types.html#access-types`’).

17.3.5.1 Stand-alone objects

```
Var        : access T := ...
Var_To_Cst : access constant T := ...
Cst        : constant access T := ...
Cst_To_Cst : constant access constant T := ...
```

In this section, we will refer to a stand-alone object of an anonymous access type as an SO.

When the restriction is in effect, the “statically deeper” relationship (see RM 3.10.2(4)) does apply to the type of a SO (contrary to RM 3.10.2(19.2)) and, for the purposes of compile-time checks, the accessibility level of the type of a SO is the accessibility level of that SO. This supports many common use-cases without the employment of `Unchecked_Access` while still removing the need for dynamic checks.

This statically disallows cases that would otherwise require a dynamic accessibility check, such as

```
type Ref is access all Integer;
Ptr : Ref;
Good : aliased Integer;

procedure Proc is
   Bad : aliased Integer;
   Stand_Alone : access Integer;
begin
   if <some condition> then
      Stand_Alone := Good'Access;
   else
      Stand_Alone := Bad'Access;
   end if;
   Ptr := Ref (Stand_Alone);
end Proc;
```

If a No_Dynamic_Accessibility_Checks restriction is in effect, then the otherwise-legal type conversion (the right-hand side of the assignment to Ptr) becomes a violation of the RM 4.6 rule “The accessibility level of the operand type shall not be statically deeper than that of the target type . . . ”.

17.3.5.2 Subprogram parameters

```
procedure P (V : access T; X : access constant T);
```

In most cases (the exceptions are described below), a No_Dynamic_Accessibility_Checks restriction means that the “statically deeper” relationship does apply to the anonymous type of an access parameter specifying an access-to-object type (contrary to RM 3.10.2(19.1)) and, for purposes of compile-time “statically deeper” checks, the accessibility level of the type of such a parameter is the accessibility level of the parameter.

This change (at least as described so far) doesn’t affect the caller’s side, but on the callee’s side it means that object designated by a non-null parameter of an anonymous access type is treated as having the same accessibility level as a local object declared immediately within the called subprogram.

With the restriction in effect, the otherwise-legal type conversion in the following example becomes illegal:

```
type Ref is access all Integer;
Ptr : Ref;

procedure Proc (Param : access Integer) is
begin
   Ptr := Ref (Param);
end Proc;
```

The aforementioned exceptions have to do with return statements from functions that either return the given parameter (in the case of a function whose result type is an anonymous access type) or return the given parameter value as an access discriminant of the function result (or of some discriminated part thereof). More specifically, the “statically deeper” changes described above do not apply for purposes of checking the “shall not be statically deeper” rule for access discriminant parts of function results (RM 6.5(5.9)) or in determining the legality of an (implicit) type conversion from the anonymous access type of a parameter of a function to an anonymous access result type of that function. In order to prevent these rule relaxations from introducing the possibility of dynamic accessibility check failures, compensating compile-time checks are performed at the call site to prevent cases where including the value of an access parameter as part of a function result could make such check failures possible (specifically, the discriminant checks of RM 6.5(21) or, in the case of an anonymous access result type, the RM 4.6(48) check performed when converting to that result type). These compile-time checks are described in the next section.

From the callee’s perspective, the level of anonymous access formal parameters would be between the level of the subprogram and the level of the subprogram’s locals. This has the effect of formal parameters being treated as local to the callee except in:

* Use as a function result
* Use as a value for an access discriminant in result object
* Use as an assignments between formal parameters

Note that with these more restricted rules we lose track of accessibility levels when assigned to local objects thus making (in the example below) the assignment to Node2.Link from Temp below compile-time illegal.

```
type Node is record
   Data : Integer;
   Link : access Node;
```

```
end record;

procedure Swap_Links (Node1, Node2 : in out Node) is
   Temp : constant access Node := Node1.Link; -- We lose the "association" to Node1
begin
   Node1.Link := Node2.Link; -- Allowed
   Node2.Link := Temp;       -- Not allowed
end;

function Identity (N : access Node) return access Node is
   Local : constant access Node := N;
begin
   if True then
      return N;              -- Allowed
   else
      return Local;          -- Not allowed
   end if;
end;
```

17.3.5.3 Function results

```
function Get (X : Rec) return access T;
```

If the result subtype of a function is either an anonymous access (sub)type, a class-wide (sub)type, an unconstrained subtype with an access discriminant, or a type with an unconstrained subcomponent subtype that has at least one access discriminant (this last case is only possible if the access discriminant has a default value), then we say that the function result type "might require an anonymous-access-part accessibility check". If a function has an access parameter, or a parameter whose subtype "might require an anonymous-access-part accessibility check", then we say that the each such parameter "might be used to pass in an anonymous-access value". If the first of these conditions holds for the result subtype of a function and the second condition holds for at least one parameter that function, then it is possible that a call to that function could return a result that contains anonymous-access values that were passed in via the parameter.

Given a function call where the result type "might require an anonymous-access-part accessibility check" and a formal parameter of that function that "might be used to pass in an anonymous-access value", either the type of that formal parameter is an anonymous access type or it is not. If it is, and if a No_Dynamic_Access_Checks restriction is in effect, then the accessibility level of the type of the actual parameter shall be statically known to not be deeper than that of the master of the call. If it isn't, then the accessibility level of the actual parameter shall be statically known to not be deeper than that of the master of the call.

Function result example:

```
declare
   type T is record
      Comp : aliased Integer;
   end record;
```

```
   function Identity (Param : access Integer) return access Integer is
   begin
      return Param;        -- Legal
   end;

   function Identity_2 (Param : aliased Integer) return access Integer is
   begin
      return Param'Access; -- Legal
   end;

   X : access Integer;
begin
   X := Identity (X);      -- Legal
   declare
      Y : access Integer;
      Z : aliased Integer;
   begin
      X := Identity (Y);   -- Illegal since Y is too deep
      X := Identity_2 (Z); -- Illegal since Z is too deep
   end;
end;
```

In order to avoid having to expand the definition of "might be used to pass in an anonymous-access value" to include any parameter of a tagged type, the No_Dynamic_Access_Checks restriction also imposes a requirement that a type extension cannot include the explicit definition of an access discriminant.

Here is an example of one such case of an upward conversion which would lead to a memory leak:

```
declare
   type T is tagged null record;
   type T2 (Disc : access Integer) is new T with null record; -- Must be illegal

   function Identity (Param : aliased T'Class) return access Integer is
   begin
      return T2 (T'Class (Param)).Disc; -- Here P gets effectively returned and set to X
   end;

   X : access Integer;
begin
   declare
      P : aliased Integer;
      Y : T2 (P'Access);
   begin
      X := Identity (T'Class (Y)); -- Pass local variable P (via Y's discriminant),
                                   -- leading to a memory leak.
   end;
end;
```

````
```

Thus we need to make the following illegal to avoid such situations:

```ada
package Pkg1 is
   type T1 is tagged null record;
   function Func (X1 : T1) return access Integer is (null);
end;

package Pkg2 is
   type T2 (Ptr1, Ptr2 : access Integer) is new Pkg1.T1 with null record; -- Illegal
   ...
end;
````

In order to prevent upward conversions of anonymous function results (like below), we also would need to assure that the level of such a result (from the callee's perspective) is statically deeper:

```
declare
   type Ref is access all Integer;
   Ptr : Ref;
   function Foo (Param : access Integer) return access Integer is
   begin
      return Result : access Integer := Param; do
         Ptr := Ref (Result); -- Not allowed
      end return;
   end;
begin
   declare
      Local : aliased Integer;
   begin
      Foo (Local'Access).all := 123;
   end;
end;
```

17.3.6 Case pattern matching

The selector for a case statement (but not for a case expression) may be of a composite type, subject to some restrictions (described below). Aggregate syntax is used for choices of such a case statement; however, in cases where a "normal" aggregate would require a discrete value, a discrete subtype may be used instead; box notation can also be used to match all values.

Consider this example:

```
type Rec is record
   F1, F2 : Integer;
end record;

procedure Caser_1 (X : Rec) is
```

```
begin
   case X is
      when (F1 => Positive, F2 => Positive) =>
         Do_This;
      when (F1 => Natural, F2 => <>) | (F1 => <>, F2 => Natural) =>
         Do_That;
      when others =>
          Do_The_Other_Thing;
   end case;
end Caser_1;
```

If `Caser_1` is called and both components of X are positive, then `Do_This` will be called; otherwise, if either component is nonnegative then `Do_That` will be called; otherwise, `Do_The_Other_Thing` will be called.

In addition, pattern bindings are supported. This is a mechanism for binding a name to a component of a matching value for use within an alternative of a case statement. For a component association that occurs within a case choice, the expression may be followed by `is <identifier>`. In the special case of a "box" component association, the identifier may instead be provided within the box. Either of these indicates that the given identifier denotes (a constant view of) the matching subcomponent of the case selector.

> **Attention:** Binding is not yet supported for arrays or subcomponents thereof.

Consider this example (which uses type `Rec` from the previous example):

```
procedure Caser_2 (X : Rec) is
begin
   case X is
      when (F1 => Positive is Abc, F2 => Positive) =>
         Do_This (Abc)
      when (F1 => Natural is N1, F2 => <N2>) |
           (F1 => <N2>, F2 => Natural is N1) =>
         Do_That (Param_1 => N1, Param_2 => N2);
      when others =>
         Do_The_Other_Thing;
   end case;
end Caser_2;
```

This example is the same as the previous one with respect to determining whether `Do_This`, `Do_That`, or `Do_The_Other_Thing` will be called. But for this version, `Do_This` takes a parameter and `Do_That` takes two parameters. If `Do_This` is called, the actual parameter in the call will be `X.F1`.

If `Do_That` is called, the situation is more complex because there are two choices for that alternative. If `Do_That` is called because the first choice matched (i.e., because `X.F1` is nonnegative and either `X.F1` or `X.F2` is zero or negative), then the actual parameters of the call will be (in order) `X.F1` and `X.F2`. If `Do_That` is called because the second choice matched (and the first one did not), then the actual parameters will be reversed.

Within the choice list for single alternative, each choice must define the same set of bindings and the component subtypes for for a given identifier must all statically match. Currently, the case of a binding for a nondiscrete component is not implemented.

If the set of values that match the choice(s) of an earlier alternative overlaps the corresponding set of a later alternative, then the first set shall be a proper subset of the second (and the later alternative will not be executed if the earlier alternative “matches”). All possible values of the composite type shall be covered. The composite type of the selector shall be an array or record type that is neither limited nor class-wide. Currently, a “when others =>” case choice is required; it is intended that this requirement will be relaxed at some point.

If a subcomponent’s subtype does not meet certain restrictions, then the only value that can be specified for that subcomponent in a case choice expression is a “box” component association (which matches all possible values for the subcomponent). This restriction applies if:

- the component subtype is not a record, array, or discrete type; or
- the component subtype is subject to a non-static constraint or has a predicate; or:
- the component type is an enumeration type that is subject to an enumeration representation clause; or
- the component type is a multidimensional array type or an array type with a nonstatic index subtype.

Support for casing on arrays (and on records that contain arrays) is currently subject to some restrictions. Non-positional array aggregates are not supported as (or within) case choices. Likewise for array type and subtype names. The current implementation exceeds compile-time capacity limits in some annoyingly common scenarios; the message generated in such cases is usually “Capacity exceeded in compiling case statement with composite selector type”.

17.3.7 Mutably Tagged Types with Size’Class Aspect

For a specific tagged nonformal type T that satisfies some conditions described later in this section, the universal-integer-valued type-related representation aspect `Size'Class` may be specified; any such specified aspect value shall be static.

Specifying this aspect imposes an upper bound on the sizes of all specific descendants of T (including T itself). T’Class (but not T) is then said to be a “mutably tagged” type - meaning that T’Class is a definite subtype and that the tag of a variable of type T’Class may be modified by assignment in some cases described later in this section. An inherited `Size'Class` aspect value may be overridden, but not with a larger value.

If the `Size'Class` aspect is specified for a type T, then every specific descendant of T (including T itself)

* shall have a Size that does not exceed the specified value; and
* shall have a (possibly inherited) `Size'Class` aspect that does not exceed the specifed value; and
* shall be undiscriminated; and
* shall have no composite subcomponent whose subtype is subject to a nonstatic constraint; and

* shall not have a tagged partial view other than a private extension; and
* shall not be a descendant of an interface type; and
* shall not have a statically deeper accessibility level than that of T.

If the `Size'Class` aspect is not specified for a type T (either explicitly or by inheritance), then it shall not be specified for any descendant of T.

Example:

```
type Root_Type is tagged null record with Size'Class => 16 * 8;

type Derived_Type is new Root_Type with record
   Stuff : Some_Type;
end record;  -- ERROR if Derived_Type exceeds 16 bytes
```

Because any subtype of a mutably tagged type is definite, it can be used as a component subtype for enclosing array or record types, as the subtype of a default-initialized stand-alone object, or as the subtype of an uninitialized allocator, as in this example:

```
Obj : Root_Type'Class;
type Array_of_Roots is array (Positive range <>) of Root_Type'Class;
```

Default initialization of an object of such a definite subtype proceeds as for the corresponding specific type, except that Program_Error is raised if the specific type is abstract. In particular, the initial tag of the object is that of the corresponding specific type.

There is a general design principle that if a type has a tagged partial view, then the type's `Size'Class` aspect (or lack thereof) should be determinable by looking only at the partial view. That provides the motivation for the rules of the next two paragraphs.

If a type has a tagged partial view, then a `Size'Class` aspect specification may be provided only at the point of the partial view declaration (in other words, no such aspect specification may be provided when the full view of the type is declared). All of the above rules (in particular, the rule that an overriding `Size'Class` aspect value shall not be larger than the overridden inherited value) are also enforced when the full view (which may have a different ancestor type than that of the partial view) is declared. If a partial view for a type inherits a `Size'Class` aspect value and does not override that value with an explicit aspect specification, then the (static) aspect values inherited by the partial view and by the full view shall be equal.

An actual parameter of an instantiation whose corresponding formal parameter is a formal tagged private type shall not be either mutably tagged or the corresponding specific type of a mutably tagged type.

For the legality rules in this section, the RM 12.3(11) rule about legality checking in the visible part and formal part of an instance is extended (in the same way that it is extended in many other places in the RM) to include the private part of an instance.

An object (or a view thereof) of a tagged type is defined to be "tag-constrained" if it is

* an object whose type is not mutably tagged; or
* a constant object; or
* a view conversion of a tag-constrained object; or
* a view conversion to a type that is not a descendant of the operand's type; or

* a formal in out or out parameter whose corresponding actual parameter is tag-constrained; or
* a dereference of an access value.

In the case of an assignment to a tagged variable that is not tag-constrained, no check is performed that the tag of the value of the expression is the same as that of the target (RM 5.2 notwithstanding). Instead, the tag of the target object becomes that of the source object of the assignment. Note that the tag of an object of a mutably tagged type MT will always be the tag of some specific type that is a descendant of MT. An assignment to a composite object similarly copies the tags of any subcomponents of the source object that have a mutably tagged type.

The Constrained attribute is defined for any name denoting an object of a mutably tagged type (RM 3.7.2 notwithstanding). In this case, the Constrained attribute yields the value True if the object is tag-constrained and False otherwise.

Renaming is not allowed (see RM 8.5.1) for a type conversion having an operand of a mutably tagged type MT and a target type TT such that TT (or its corresponding specific type if TT is class-wide) is not an ancestor of MT (this is sometimes called a "downward" conversion), nor for any part of such an object, nor for any slice of any part of such an object. This rule also applies in any context where a name is required to be one for which "renaming is allowed" (for example, see RM 12.4). [This is analogous to the way that renaming is not allowed for a discriminant-dependent component of an unconstrained variable.]

A name denoting a view of a variable of a mutably tagged type shall not occur as an operative constituent of the prefix of a name denoting a prefixed view of a callable entity, except as the callee name in a call to the callable entity. This disallows, for example, renaming such a prefixed view, passing the prefixed view name as a generic actual parameter, or using the prefixed view name as the prefix of an attribute.

The execution of a construct is erroneous if the construct has a constituent that is a name denoting a subcomponent of a tagged object and the object's tag is changed by this execution between evaluating the name and the last use (within this execution) of the subcomponent denoted by the name. This is analogous to the RM 3.7.2(4) rule about discriminant-dependent subcomponents.

If the type of a formal parameter is a specific tagged type, then the execution of the call is erroneous if the tag of the actual is changed while the formal parameter exists (that is, before leaving the corresponding callable construct). This is analogous to the RM 6.4.1(18) rule about discriminated parameters.

17.3.8 No_Raise aspect

The `No_Raise` aspect can be applied to a subprogram to declare that this subprogram is not expected to raise an exception. Should an exception still be raised during the execution of the subprogram, it is caught at the end of this execution and `Program_Error` is propagated to the caller.

17.3.9 Inference of Dependent Types in Generic Instantiations

If a generic formal type T2 depends on another formal type T1, the actual for T1 can be inferred from the actual for T2. That is, you can give the actual for T2, and leave out the one for T1.

For example, `Ada.Unchecked_Deallocation` has two generic formals:

```
generic
   type Object (<>) is limited private;
   type Name is access Object;
procedure Ada.Unchecked_Deallocation (X : in out Name);
```

where `Name` depends on `Object`. With this language extension, you can leave out the actual for `Object`, as in:

```
type Integer_Access is access all Integer;

procedure Free is new Unchecked_Deallocation (Name => Integer_Access);
```

The compiler will infer that the actual type for `Object` is `Integer`. Note that named notation is always required when using inference.

The following inferences are allowed:

- For a formal access type, the designated type can be inferred.
- For a formal array type, the index type(s) and the component type can be inferred.
- For a formal type with discriminants, the type(s) of the discriminants can be inferred.

Example for arrays:

```
generic
   type Element_Type is private;
   type Index_Type is (<>);
   type Array_Type is array (Index_Type range <>) of Element_Type;
package Array_Operations is
   ...
end Array_Operations;

...

type Int_Array is array (Positive range <>) of Integer;

package Int_Array_Operations is new Array_Operations (Array_Type => Int_Array);
```

The index and component types of `Array_Type` are inferred from `Int_Array`, so that the above instantiation is equivalent to the following standard-Ada instantiation:

```
package Int_Array_Operations is new Array_Operations
   (Element_Type => Integer,
      Index_Type   => Positive,
      Array_Type   => Int_Array);
```

17.3.10 External_Initialization Aspect

The `External_Initialization` aspect provides a feature similar to Rust's `include_bytes!` and to C23's `#embed`. It has the effect of initializing an object with the contents of a file specified by a file path.

Only string objects and objects of type `Ada.Streams.Stream_Element_Array` can be subject to the `External_Initialization` aspect.

Example:

```
with Ada.Streams;

package P is
   S : constant String with External_Initialization => "foo.txt";

   X : constant Ada.Streams.Stream_Element_Array with External_Initialization => "bar.bin";
end P;
```

`External_Initialization` aspect accepts the following parameters:

- mandatory `Path`: the path the compiler uses to access the binary resource.

If `Path` is a relative path, it is interpreted relatively to the directory of the file that contains the aspect specification.

> **Attention:** The maximum size of loaded files is limited to 2^31 bytes.

17.3.11 Finally construct

The `finally` keyword makes it possible to have a sequence of statements be executed when another sequence of statements is completed, whether normally or abnormally.

This feature is similar to the one with the same name in other languages such as Java.

17.3.11.1 Syntax

```
handled_sequence_of_statements ::=
     sequence_of_statements
  [exception
     exception_handler
    {exception_handler}]
  [finally
    sequence_of_statements]
```

17.3.11.2 Legality Rules

Return statements in the `sequence_of_statements` attached to the finally that would cause control to be transferred outside the finally part are forbidden.

Goto & exit where the target is outside of the finally's `sequence_of_statements` are forbidden

17.3.11.3 Dynamic Semantics

Statements in the optional `sequence_of_statements` contained in the `finally` branch will be executed unconditionally, after the main `sequence_of_statements` is executed, and after any potential `exception_handler` is executed.

If an exception is raised in the finally part, it cannot be caught by the `exception_handler`.

Abort/ATC (asynchronous transfer of control) cannot interrupt a finally block, nor prevent its execution, that is the finally block must be executed in full even if the containing task is aborted, or if the control is transferred out of the block.

17.3.12 Continue statement

The `continue` keyword makes it possible to stop execution of a loop iteration and continue with the next one. A continue statement has the same syntax (except "exit" is replaced with "continue"), static semantics, and legality rules as an exit statement. The difference is in the dynamic semantics: where an exit statement would cause a transfer of control that completes the (implicitly or explicitly) specified loop_statement, a continue statement would instead cause a transfer of control that completes only the current iteration of that loop_statement, like a goto statement targeting a label following the last statement in the sequence of statements of the specified loop_statement.

Note that `continue` is a keyword but it is not a reserved word. This is a configuration that does not exist in standard Ada.

17.3.13 Destructors

The `Destructor` extension adds a new finalization mechanism that significantly differs standard Ada in how it interacts with type derivation.

New syntax is introduced to make it possible to define "destructors" for record types, tagged or untagged. Here's a simple example:

```
package P is
   type T is record
      ...
   end record;

   procedure T'Destructor (X : in out T);
end P;
package body P is
   procedure T'Destructor (X : in out T) is
   begin
      ...
   end T'Destructor;
end P;
```

Like `Finalize` procedures, destructors are called on objects just before they are destroyed. But destructors are more flexible in how they can used with derived types. With standard Ada finalization, when you derive from a finalizable type, you must either inherit the `Finalize` procedure or override it completely.

Destructors work differently. You can define a destructor for a type derived from a parent type that also has a destructor, and then when objects of the derived type are finalized, both destructors will be called. For example:

```
type T1 is record
   ...
end record;

procedure T1'Destructor (X : in out T1);

type T2 is new T1;
```

```
procedure T2'Destructor (X : in out T2);
```

When an object of type `T2` is finalized, there will be first a call to `T2'Destructor`, and then a call to `T1'Destructor` on the object.

17.3.14 Structural Generic Instantiation

The compiler implements a second kind of generic instantiation, called "structural", alongside the traditional instantiation specified by the language, which is defined as follows: the structural instantiation of a generic unit on given actual parameters is the anonymous instantiation of the generic unit on the actual parameters done in the outermost scope where it would be legal to do an identical traditional instantiation.

There is at most one structural instantiation of a generic unit on given actual parameters done in a partition.

Structural generic instances (the product of structural instantiation) are implicitly created whenever a reference to them is made in a place where a name is accepted by the language.

17.3.14.1 Syntax

```
name ::= { set of productions specified in the RM }
       | structural_generic_instance_name

structural_generic_instance_name ::= name generic_actual_part
```

17.3.14.2 Legality Rules

The `name` in a `structural_generic_instance_name` shall denote a generic unit that is preelaborated. Note that, unlike in a traditional instantiation, there are no square brackets around the `generic_actual_part` in the second production, which means that it is mandatory and, therefore, that the generic unit shall have at least one generic formal parameter.

The generic unit shall not take a generic formal object of mode `in out`. If the generic unit takes a generic formal object of mode `in`, then the corresponding generic actual parameter shall be a static expression.

A `structural_generic_instance_name` for a generic package shall not be present in a library unit if the structural instance is also a library unit and has a semantic dependence on the former.

For a generic subprogram, if a local entity of the enclosing library-level package is used as an actual and the structural instance would have a semantic dependence on the package, the structural instantiation is automatically demoted to a local instantiation. In this case, several instances of the generic subprogram may be present in a single partition, unless whole-partition optimization is performed (e.g., via LTO).

17.3.14.3 Static Semantics

A `structural_generic_instance_name` denotes the instance that is the product of the structural instantiation of a generic unit on the specified actual parameters. This instance is unique to a partition, except when a generic subprogram instantiation is automatically demoted to a local instantiation as described under Legality Rules.

Example:

```
with Ada.Containers.Vectors;
```

```
procedure P is
   V : Ada.Containers.Vectors(Positive,Integer).Vector;

begin
   V.Append (1);
   V.Append (0);
   Ada.Containers.Vectors(Positive,Integer).Generic_Sorting("<").Sort (V);
end;
```

This procedure references two structural instantiations of two different generic units: `Ada.Containers.Vectors(Positive,Integer)` is the structural instance of the generic unit `Ada.Containers.Vectors` on `Positive` and `Integer` and `Ada.Containers.Vectors(Positive,Integer).Generic_Sorting("<")` is the structural instance of the nested generic unit `Ada.Containers.Vectors(Positive,Integer).Generic_Sorting` on `"<"`.

Note that the following example is illegal:

```
with Ada.Containers.Vectors;

package Q is
   type T is record
      I : Integer;
   end record;

   V : Ada.Containers.Vectors(Positive,T).Vector;
end Q;
```

The reason is that `Ada.Containers.Vectors`, `Positive` and `Q.T` being library-level entities, the structural instance `Ada.Containers.Vectors(Positive,T)` is a library unit with a dependence on `Q` and, therefore, cannot be referenced from within `Q`. This restriction applies to structural instantiations of generic packages. The simple way out is to declare a traditional instantiation in this case:

```
with Ada.Containers.Vectors;

package Q is
   type T is record
      I : Integer;
   end record;

   package Vectors_Of_T is new Ada.Containers.Vectors(Positive,T);

   V : Vectors_Of_T.Vector;
end Q;
```

But the following example is legal:

```
with Ada.Containers.Vectors;

procedure P is
```

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

   V : Ada.Containers.Vectors(Positive,T).Vector;
end;
```

because the structural instance `Ada.Containers.Vectors(Positive,T)` is not a library unit.

For generic subprograms, the restriction does not apply: if a local entity of a library-level package is used as an actual, the structural instantiation is automatically demoted to a local instantiation. For example:

```
with Ada.Unchecked_Deallocation;

package Q is
   type T is record
      I : Integer;
   end record;

   type T_Access is access T;
end Q;

package body Q is
   procedure Free_T (X : in out T_Access) is
   begin
      Ada.Unchecked_Deallocation(T, T_Access)(X);
   end Free_T;
end Q;
```

is legal: since `T` and `T_Access` are local entities of `Q`, the structural instantiation of `Ada.Unchecked_Deallocation` is demoted to a local instantiation rather than producing an error. Note that the uniqueness guarantee no longer holds in this case and several instances of the generic subprogram may be present in a single partition.

The first example can be rewritten in a less verbose manner:

```
with Ada.Containers.Vectors; use Ada.Containers.Vectors(Positive,Integer);

procedure P is
   V : Vector;

begin
   V.Append (1);
   V.Append (0);
   Generic_Sorting("<").Sort (V);
end;
```

Another example, which additionally uses the inference of dependent types:

```
with Ada.Unchecked_Deallocation;
```

```
procedure P is

   type Integer_Access is access all Integer;

   A : Integer_Access := new Integer'(1);

begin
   Ada.Unchecked_Deallocation(Name => Integer_Access) (A);
end;
```

18 Security Hardening Features

This chapter describes Ada extensions aimed at security hardening that are provided by GNAT.

The features in this chapter are currently experimental and subject to change.

These features are supported only by the GCC back end, not by LLVM.

18.1 Register Scrubbing

GNAT can generate code to zero-out hardware registers before returning from a subprogram.

It can be enabled with the `-fzero-call-used-regs=`choice'` command-line option, to affect all subprograms in a compilation, and with a `Machine_Attribute` pragma, to affect only specific subprograms.

```
procedure Foo;
pragma Machine_Attribute (Foo, "zero_call_used_regs", "used");
--  Before returning, Foo scrubs only call-clobbered registers
--  that it uses itself.

function Bar return Integer;
pragma Machine_Attribute (Bar, "zero_call_used_regs", "all");
--  Before returning, Bar scrubs all call-clobbered registers.

function Baz return Integer;
pragma Machine_Attribute (Bar, "zero_call_used_regs", "leafy");
--  Before returning, Bar scrubs call-clobbered registers, either
--  those it uses itself, if it can be identified as a leaf
--  function, or all of them otherwise.
```

For usage and more details on the command-line option, on the `zero_call_used_regs` attribute, and on their use with other programming languages, see *Using the GNU Compiler Collection (GCC)*.

18.2 Stack Scrubbing

GNAT can generate code to zero-out stack frames used by subprograms.

It can be activated with the `Machine_Attribute` pragma, on specific subprograms and variables, or their types. (This attribute always applies to a type, even when it is associated with a subprogram or a variable.)

```
function Foo returns Integer;
pragma Machine_Attribute (Foo, "strub");
--  Foo and its callers are modified so as to scrub the stack
--  space used by Foo after it returns.  Shorthand for:
--  pragma Machine_Attribute (Foo, "strub", "at-calls");

procedure Bar;
pragma Machine_Attribute (Bar, "strub", "internal");
--  Bar is turned into a wrapper for its original body,
```

```
--  and they scrub the stack used by the original body.

Var : Integer;
pragma Machine_Attribute (Var, "strub");
--  Reading from Var in a subprogram enables stack scrubbing
--  of the stack space used by the subprogram.  Furthermore, if
--  Var is declared within a subprogram, this also enables
--  scrubbing of the stack space used by that subprogram.
```

Given these declarations, Foo has its type and body modified as follows:

```
function Foo (<WaterMark> : in out System.Address) returns Integer
is
  --  ...
begin
  <__strub_update> (<WaterMark>);  --  Updates the stack WaterMark.
  --  ...
end;
```

whereas its callers are modified from:

```
X := Foo;
```

to:

```
declare
  <WaterMark> : System.Address;
begin
  <__strub_enter> (<WaterMark>);  -- Initialize <WaterMark>.
  X := Foo (<WaterMark>);
  <__strub_leave> (<WaterMark>);  -- Scrubs stack up to <WaterMark>.
end;
```

As for Bar, because it is strubbed in internal mode, its callers are not modified. Its definition is modified roughly as follows:

```
procedure Bar is
  <WaterMark> : System.Address;
  procedure Strubbed_Bar (<WaterMark> : in out System.Address) is
  begin
    <__strub_update> (<WaterMark>);  --  Updates the stack WaterMark.
    -- original Bar body.
  end Strubbed_Bar;
begin
  <__strub_enter> (<WaterMark>);  -- Initialize <WaterMark>.
  Strubbed_Bar (<WaterMark>);
  <__strub_leave> (<WaterMark>);  -- Scrubs stack up to <WaterMark>.
end Bar;
```

There are also `-fstrub=`choice'` command-line options to control default settings. For usage and more details on the command-line options, on the `strub` attribute, and their use with other programming languages, see *Using the GNU Compiler Collection (GCC)*.

Note that Ada secondary stacks are not scrubbed. The restriction `No_Secondary_Stack` avoids their use, and thus their accidental preservation of data that should be scrubbed.

Attributes `Access` and `Unconstrained_Access` of variables and constants with `strub` enabled require types with `strub` enabled; there is no way to express an access-to-strub type otherwise. `Unchecked_Access` bypasses this constraint, but the resulting access type designates a non-strub type.

```
VI : aliased Integer;
pragma Machine_Attribute (VI, "strub");
XsVI : access Integer := VI'Access; -- Error.
UXsVI : access Integer := VI'Unchecked_Access; -- OK,
--  UXsVI does *not* enable strub in subprograms that
--  dereference it to obtain the UXsVI.all value.

type Strub_Int is new Integer;
pragma Machine_Attribute (Strub_Int, "strub");
VSI : aliased Strub_Int;
XsVSI : access Strub_Int := VSI'Access; -- OK,
--  VSI and XsVSI.all both enable strub in subprograms that
--  read their values.
```

Every access-to-subprogram type, renaming, and overriding and overridden dispatching operations that may refer to a subprogram with an attribute-modified interface must be annotated with the same interface-modifying attribute. Access-to-subprogram types can be explicitly converted to different strub modes, as long as they are interface-compatible (i.e., adding or removing `at-calls` is not allowed). For example, a `strub-disabled` subprogram can be turned `callable` through such an explicit conversion:

```
type TBar is access procedure;

type TBar_Callable is access procedure;
pragma Machine_Attribute (TBar_Callable, "strub", "callable");
--  The attribute modifies the procedure type, rather than the
--  access type, because of the extra argument after "strub",
--  only applicable to subprogram types.

Bar_Callable_Ptr : constant TBar_Callable
           := TBar_Callable (TBar'(Bar'Access));

procedure Bar_Callable renames Bar_Callable_Ptr.all;
pragma Machine_Attribute (Bar_Callable, "strub", "callable");
```

Note that the renaming declaration is expanded to a full subprogram body, it won't be just an alias. Only if it is inlined will it be as efficient as a call by dereferencing the access-to-subprogram constant Bar_Callable_Ptr.

18.3 Hardened Conditionals

GNAT can harden conditionals to protect against control-flow attacks.

This is accomplished by two complementary transformations, each activated by a separate command-line option.

The option `-fharden-compares` enables hardening of compares that compute results stored in variables, adding verification that the reversed compare yields the opposite result, turning:

```
B := X = Y;
```

into:

```
B := X = Y;
declare
  NotB : Boolean := X /= Y; -- Computed independently of B.
begin
  if B = NotB then
    <__builtin_trap>;
  end if;
end;
```

The option `-fharden-conditional-branches` enables hardening of compares that guard conditional branches, adding verification of the reversed compare to both execution paths, turning:

```
if X = Y then
  X := Z + 1;
else
  Y := Z - 1;
end if;
```

into:

```
if X = Y then
  if X /= Y then -- Computed independently of X = Y.
    <__builtin_trap>;
  end if;
  X := Z + 1;
else
  if X /= Y then -- Computed independently of X = Y.
    null;
  else
    <__builtin_trap>;
  end if;
  Y := Z - 1;
end if;
```

These transformations are introduced late in the compilation pipeline, long after boolean expressions are decomposed into separate compares, each one turned into either a conditional branch or a compare whose result is stored in a boolean variable or temporary. Compiler optimizations, if enabled, may also turn conditional branches into stored compares, and vice-versa, or into operations with implied conditionals (e.g. MIN and MAX). Conditionals may also be optimized out entirely, if their value can be determined at compile time, and occasionally multiple compares can be combined into one.

It is thus difficult to predict which of these two options will affect a specific compare operation expressed in source code. Using both options ensures that every compare that is neither optimized out nor optimized into implied conditionals will be hardened.

The addition of reversed compares can be observed by enabling the dump files of the corresponding passes, through command-line options `-fdump-tree-hardcmp` and `-fdump-tree-hardcbr`, respectively.

They are separate options, however, because of the significantly different performance impact of the hardening transformations.

For usage and more details on the command-line options, see *Using the GNU Compiler Collection (GCC)*. These options can be used with other programming languages supported by GCC.

18.4 Hardened Booleans

Ada has built-in support for introducing boolean types with alternative representations, using representation clauses:

```
type HBool is new Boolean;
for HBool use (16#5a#, 16#a5#);
for HBool'Size use 8;
```

When validity checking is enabled, the compiler will check that variables of such types hold values corresponding to the selected representations.

There are multiple strategies for where to introduce validity checking (see `-gnatV` options). Their goal is to guard against various kinds of programming errors, and GNAT strives to omit checks when program logic rules out an invalid value, and optimizers may further remove checks found to be redundant.

For additional hardening, the `hardbool` `Machine_Attribute` pragma can be used to annotate boolean types with representation clauses, so that expressions of such types used as conditions are checked even when compiling with `-gnatVT`:

```
pragma Machine_Attribute (HBool, "hardbool");

function To_Boolean (X : HBool) returns Boolean is (Boolean (X));
```

is compiled roughly like:

```
function To_Boolean (X : HBool) returns Boolean is
begin
  if X not in True | False then
    raise Constraint_Error;
  elsif X in True then
    return True;
  else
    return False;
  end if;
end To_Boolean;
```

Note that `-gnatVn` will disable even `hardbool` testing.

Analogous behavior is available as a GCC extension to the C and Objective C programming languages, through the `hardbool` attribute, with the difference that, instead of raising a Constraint_Error exception, when a hardened boolean variable is found to hold a value that stands for neither True nor False, the program traps. For usage and more details on that attribute, see *Using the GNU Compiler Collection (GCC)*.

18.5 Control Flow Redundancy

GNAT can guard against unexpected execution flows, such as branching into the middle of subprograms, as in Return Oriented Programming exploits.

In units compiled with `-fharden-control-flow-redundancy`, subprograms are instrumented so that, every time they are called, basic blocks take note as control flows through them, and, before returning, subprograms verify that the taken notes are consistent with the control-flow graph.

The performance impact of verification on leaf subprograms can be much higher, while the averted risks are much lower on them. Instrumentation can be disabled for leaf subprograms with `-fhardcfr-skip-leaf`.

Functions with too many basic blocks, or with multiple return points, call a run-time function to perform the verification. Other functions perform the verification inline before returning.

Optimizing the inlined verification can be quite time consuming, so the default upper limit for the inline mode is set at 16 blocks. Command-line option `--param hardcfr-max-inline-blocks=` can override it.

Even though typically sparse control-flow graphs exhibit run-time verification time nearly proportional to the block count of a subprogram, it may become very significant for generated subprograms with thousands of blocks. Command-line option `--param hardcfr-max-blocks=` can set an upper limit for instrumentation.

For each block that is marked as visited, the mechanism checks that at least one of its predecessors, and at least one of its successors, are also marked as visited.

Verification is performed just before a subprogram returns. The following fragment:

```
if X then
  Y := F (Z);
  return;
end if;
```

gets turned into:

```
type Visited_Bitmap is array (1..N) of Boolean with Pack;
Visited : aliased Visited_Bitmap := (others => False);
--  Bitmap of visited blocks.  N is the basic block count.
[...]
--  Basic block #I
Visited(I) := True;
if X then
  --  Basic block #J
  Visited(J) := True;
  Y := F (Z);
  CFR.Check (N, Visited'Access, CFG'Access);
  --  CFR is a hypothetical package whose Check procedure calls
  --  libgcc's __hardcfr_check, that traps if the Visited bitmap
  --  does not hold a valid path in CFG, the run-time
  --  representation of the control flow graph in the enclosing
  --  subprogram.
```

```
  return;
end if;
--  Basic block #K
Visited(K) := True;
```

Verification would also be performed before tail calls, if any front-ends marked them as mandatory or desirable, but none do. Regular calls are optimized into tail calls too late for this transformation to act on it.

In order to avoid adding verification after potential tail calls, which would prevent tail-call optimization, we recognize returning calls, i.e., calls whose result, if any, is returned by the calling subprogram to its caller immediately after the call returns. Verification is performed before such calls, whether or not they are ultimately optimized to tail calls. This behavior is enabled by default whenever sibcall optimization is enabled (see `-foptimize-sibling-calls`); it may be disabled with `-fno-hardcfr-check-returning-calls`, or enabled with `-fhardcfr-check-returning-calls`, regardless of the optimization, but the lack of other optimizations may prevent calls from being recognized as returning calls:

```
--  CFR.Check here, with -fhardcfr-check-returning-calls.
P (X);
--  CFR.Check here, with -fno-hardcfr-check-returning-calls.
return;
```

or:

```
--  CFR.Check here, with -fhardcfr-check-returning-calls.
R := F (X);
--  CFR.Check here, with -fno-hardcfr-check-returning-calls.
return R;
```

Any subprogram from which an exception may escape, i.e., that may raise or propagate an exception that isn't handled internally, is conceptually enclosed by a cleanup handler that performs verification, unless this is disabled with `-fno-hardcfr-check-exceptions`. With this feature enabled, a subprogram body containing:

```
--  ...
  Y := F (X);  -- May raise exceptions.
--  ...
  raise E;  -- Not handled internally.
--  ...
```

gets modified as follows:

```
begin
  --  ...
    Y := F (X);  -- May raise exceptions.
  --  ...
    raise E;  -- Not handled internally.
  --  ...
exception
  when others =>
    CFR.Check (N, Visited'Access, CFG'Access);
    raise;
end;
```

Verification may also be performed before No_Return calls, whether all of them, with `-fhardcfr-check-noreturn-calls=always`; all but internal subprograms involved in exception-raising or -reraising or subprograms explicitly marked with both `No_Return` and `Machine_Attribute expected_throw` pragmas, with `-fhardcfr-check-noreturn-calls=no-xthrow` (default); only nothrow ones, with `-fhardcfr-check-noreturn-calls=nothrow`; or none, with `-fhardcfr-check-noreturn-calls=never`.

When a No_Return call returns control to its caller through an exception, verification may have already been performed before the call, if `-fhardcfr-check-noreturn-calls=always` or `-fhardcfr-check-noreturn-calls=no-xthrow` is in effect. The compiler arranges for already-checked No_Return calls without a preexisting handler to bypass the implicitly-added cleanup handler and thus the redundant check, but a local exception or cleanup handler, if present, will modify the set of visited blocks, and checking will take place again when the caller reaches the next verification point, whether it is a return or reraise statement after the exception is otherwise handled, or even another No_Return call.

The instrumentation for hardening with control flow redundancy can be observed in dump files generated by the command-line option `-fdump-tree-hardcfr`.

For more details on the control flow redundancy command-line options, see *Using the GNU Compiler Collection (GCC)*. These options can be used with other programming languages supported by GCC.

19 Obsolescent Features

This chapter describes features that are provided by GNAT, but are considered obsolescent since there are other, more appropriate, ways of achieving the same effect. These features are provided solely for historical compatibility purposes.

19.1 PolyORB

PolyORB is a deprecated product. It will be baselined with the GNAT Pro release 28. After this release, there will be no new versions of this product. Contact AdaCore support to get recommendations for replacements.

19.2 pragma No_Run_Time

The pragma `No_Run_Time` is used to achieve an affect similar to the use of the "Zero Foot Print" configurable run time, but without requiring a specially configured run time. The result of using this pragma, which must be used for all units in a partition, is to restrict the use of any language features requiring run-time support code. The preferred usage is to use an appropriately configured run-time that includes just those features that are to be made accessible.

19.3 pragma Ravenscar

The pragma `Ravenscar` has exactly the same effect as pragma `Profile (Ravenscar)`. The latter usage is preferred since it is part of the new Ada 2005 standard.

19.4 pragma Restricted_Run_Time

The pragma `Restricted_Run_Time` has exactly the same effect as pragma `Profile (Restricted)`. The latter usage is preferred since the Ada 2005 pragma `Profile` is intended for this kind of implementation dependent addition.

19.5 pragma Task_Info

The functionality provided by pragma `Task_Info` is now part of the Ada language. The `CPU` aspect and the package `System.Multiprocessors` offer a less system-dependent way to specify task affinity or to query the number of processors.

Syntax

```
pragma Task_Info (EXPRESSION);
```

This pragma appears within a task definition (like pragma `Priority`) and applies to the task in which it appears. The argument must be of type `System.Task_Info.Task_Info_Type`. The `Task_Info` pragma provides system dependent control over aspects of tasking implementation, for example, the ability to map tasks to specific processors. For details on the facilities available for the version of GNAT that you are using, see the documentation in the spec of package System.Task_Info in the runtime library.

19.6 package System.Task_Info (`s-tasinf.ads`)

This package provides target dependent functionality that is used to support the `Task_Info` pragma. The predefined Ada package `System.Multiprocessors` and the `CPU` aspect now provide a standard replacement for GNAT's `Task_Info` functionality.

20 Compatibility and Porting Guide

This chapter 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.

20.1 Writing Portable Fixed-Point Declarations

The Ada Reference Manual gives an implementation freedom to choose bounds that are narrower by `Small` from the given bounds. For example, if we write

```
type F1 is delta 1.0 range -128.0 .. +128.0;
```

then the implementation is allowed to choose -128.0 .. +127.0 if it likes, but is not required to do so.

This leads to possible portability problems, so let's have a closer look at this, and figure out how to avoid these problems.

First, why does this freedom exist, and why would an implementation take advantage of it? To answer this, take a closer look at the type declaration for `F1` above. If the compiler uses the given bounds, it would need 9 bits to hold the largest positive value (and typically that means 16 bits on all machines). But if the implementation chooses the +127.0 bound then it can fit values of the type in 8 bits.

Why not make the user write +127.0 if that's what is wanted? The rationale is that if you are thinking of fixed point as a kind of 'poor man's floating-point', then you don't want to be thinking about the scaled integers that are used in its representation. Let's take another example:

```
type F2 is delta 2.0**(-15) range -1.0 .. +1.0;
```

Looking at this declaration, it seems casually as though it should fit in 16 bits, but again that extra positive value +1.0 has the scaled integer equivalent of 2**15 which is one too big for signed 16 bits. The implementation can treat this as:

```
type F2 is delta 2.0**(-15) range -1.0 .. +1.0-(2.0**(-15));
```

and the Ada language design team felt that this was too annoying to require. We don't need to debate this decision at this point, since it is well established (the rule about narrowing the ranges dates to Ada 83).

But the important point is that an implementation is not required to do this narrowing, so we have a potential portability problem. We could imagine three types of implementation:

a. those that narrow the range automatically if they can figure out that the narrower range will allow storage in a smaller machine unit,
b. those that will narrow only if forced to by a `'Size` clause, and
c. those that will never narrow.

Now if we are language theoreticians, we can imagine a fourth approach: to narrow all the time, e.g. to treat

```
type F3 is delta 1.0 range -10.0 .. +23.0;
```

as though it had been written:

```
type F3 is delta 1.0 range -9.0 .. +22.0;
```

But although technically allowed, such a behavior would be hostile and silly, and no real compiler would do this. All real compilers will fall into one of the categories (a), (b) or (c) above.

So, how do you get the compiler to do what you want? The answer is give the actual bounds you want, and then use a `'Small` clause and a `'Size` clause to absolutely pin down what the compiler does. E.g., for `F2` above, we will write:

```
My_Small : constant := 2.0**(-15);
My_First : constant := -1.0;
My_Last  : constant := +1.0 - My_Small;

type F2 is delta My_Small range My_First .. My_Last;
```

and then add

```
for F2'Small use my_Small;
for F2'Size  use 16;
```

In practice all compilers will do the same thing here and will give you what you want, so the above declarations are fully portable. If you really want to play language lawyer and guard against ludicrous behavior by the compiler you could add

```
Test1 : constant := 1 / Boolean'Pos (F2'First = My_First);
Test2 : constant := 1 / Boolean'Pos (F2'Last  = My_Last);
```

One or other or both are allowed to be illegal if the compiler is behaving in a silly manner, but at least the silly compiler will not get away with silently messing with your (very clear) intentions.

If you follow this scheme you will be guaranteed that your fixed-point types will be portable.

20.2 Compatibility with Ada 83

Ada 95 and the subsequent revisions Ada 2005, Ada 2012, Ada 2022 are highly upwards compatible with Ada 83. In particular, the design intention was that the difficulties associated with moving from Ada 83 to later versions of the standard should be no greater than those that occur when moving from one Ada 83 system to another.

However, there are a number of points at which there are minor incompatibilities. The *Ada 95 Annotated Reference Manual* contains full details of these issues as they relate to Ada 95, and should be consulted for a complete treatment. In practice the following subsections treat the most likely issues to be encountered.

20.2.1 Legal Ada 83 programs that are illegal in Ada 95

Some legal Ada 83 programs are illegal (i.e., they will fail to compile) in Ada 95 and later versions of the standard:

* ‘Character literals’

  Some uses of character literals are ambiguous. Since Ada 95 has introduced `Wide_Character` as a new predefined character type, some uses of character literals that were legal in Ada 83 are illegal in Ada 95. For example:

  ```
  for Char in 'A' .. 'Z' loop ... end loop;
  ```

The problem is that 'A' and 'Z' could be from either `Character` or `Wide_Character`. The simplest correction is to make the type explicit; e.g.:

```
for Char in Character range 'A' .. 'Z' loop ... end loop;
```

* 'New reserved words'

  The identifiers `abstract`, `aliased`, `protected`, `requeue`, `tagged`, and `until` are reserved in Ada 95. Existing Ada 83 code using any of these identifiers must be edited to use some alternative name.

* 'Freezing rules'

  The rules in Ada 95 are slightly different with regard to the point at which entities are frozen, and representation pragmas and clauses are not permitted past the freeze point. This shows up most typically in the form of an error message complaining that a representation item appears too late, and the appropriate corrective action is to move the item nearer to the declaration of the entity to which it refers.

  A particular case is that representation pragmas cannot be applied to a subprogram body. If necessary, a separate subprogram declaration must be introduced to which the pragma can be applied.

* 'Optional bodies for library packages'

  In Ada 83, a package that did not require a package body was nevertheless allowed to have one. This lead to certain surprises in compiling large systems (situations in which the body could be unexpectedly ignored by the binder). In Ada 95, if a package does not require a body then it is not permitted to have a body. To fix this problem, simply remove a redundant body if it is empty, or, if it is non-empty, introduce a dummy declaration into the spec that makes the body required. One approach is to add a private part to the package declaration (if necessary), and define a parameterless procedure called `Requires_Body`, which must then be given a dummy procedure body in the package body, which then becomes required. Another approach (assuming that this does not introduce elaboration circularities) is to add an `Elaborate_Body` pragma to the package spec, since one effect of this pragma is to require the presence of a package body.

* 'Numeric_Error is the same exception as Constraint_Error'

  In Ada 95, the exception `Numeric_Error` is a renaming of `Constraint_Error`. This means that it is illegal to have separate exception handlers for the two exceptions. The fix is simply to remove the handler for the `Numeric_Error` case (since even in Ada 83, a compiler was free to raise `Constraint_Error` in place of `Numeric_Error` in all cases).

* 'Indefinite subtypes in generics'

  In Ada 83, it was permissible to pass an indefinite type (e.g, `String`) as the actual for a generic formal private type, but then the instantiation would be illegal if there were any instances of declarations of variables of this type in the generic body. In Ada 95, to avoid this clear violation of the methodological principle known as the 'contract model', the generic declaration explicitly indicates whether or not such instantiations are permitted. If a generic formal parameter has explicit unknown discriminants, indicated by using `(<>)` after the subtype name, then it can be instantiated with indefinite types, but no stand-alone variables can be declared of this type. Any attempt to declare such a variable will result in an illegality at the time the generic is declared. If the `(<>)`

notation is not used, then it is illegal to instantiate the generic with an indefinite type. This is the potential incompatibility issue when porting Ada 83 code to Ada 95. It will show up as a compile time error, and the fix is usually simply to add the `(<>)` to the generic declaration.

20.2.2 More deterministic semantics

* 'Conversions'

  Conversions from real types to integer types round away from 0. In Ada 83 the conversion Integer(2.5) could deliver either 2 or 3 as its value. This implementation freedom was intended to support unbiased rounding in statistical applications, but in practice it interfered with portability. In Ada 95 the conversion semantics are unambiguous, and rounding away from 0 is required. Numeric code may be affected by this change in semantics. Note, though, that this issue is no worse than already existed in Ada 83 when porting code from one vendor to another.

* 'Tasking'

  The Real-Time Annex introduces a set of policies that define the behavior of features that were implementation dependent in Ada 83, such as the order in which open select branches are executed.

20.2.3 Changed semantics

The worst kind of incompatibility is one where a program that is legal in Ada 83 is also legal in Ada 95 but can have an effect in Ada 95 that was not possible in Ada 83. Fortunately this is extremely rare, but the one situation that you should be alert to is the change in the predefined type `Character` from 7-bit ASCII to 8-bit Latin-1.

* 'Range of type ''Character''

  The range of `Standard.Character` is now the full 256 characters of Latin-1, whereas in most Ada 83 implementations it was restricted to 128 characters. Although some of the effects of this change will be manifest in compile-time rejection of legal Ada 83 programs it is possible for a working Ada 83 program to have a different effect in Ada 95, one that was not permitted in Ada 83. As an example, the expression `Character'Pos(Character'Last)` returned `127` in Ada 83 and now delivers `255` as its value. In general, you should look at the logic of any character-processing Ada 83 program and see whether it needs to be adapted to work correctly with Latin-1. Note that the predefined Ada 95 API has a character handling package that may be relevant if code needs to be adapted to account for the additional Latin-1 elements. The desirable fix is to modify the program to accommodate the full character set, but in some cases it may be convenient to define a subtype or derived type of Character that covers only the restricted range.

20.2.4 Other language compatibility issues

* '-gnat83' switch

  All implementations of GNAT provide a switch that causes GNAT to operate in Ada 83 mode. In this mode, some but not all compatibility problems of the type described above are handled automatically. For example, the new reserved words introduced in Ada 95 and Ada 2005 are treated simply as identifiers as in Ada 83. However, in practice, it is usually advisable to make the necessary modifications to the program to remove the need for using this switch. See the `Compiling Different Versions of Ada` section in the *GNAT User's Guide*.

* Support for removed Ada 83 pragmas and attributes

  A number of pragmas and attributes from Ada 83 were removed from Ada 95, generally because they were replaced by other mechanisms. Ada 95 and Ada 2005 compilers are allowed, but not required, to implement these missing elements. In contrast with some other compilers, GNAT implements all such pragmas and attributes, eliminating this compatibility concern. These include `pragma Interface` and the floating point type attributes (`Emax`, `Mantissa`, etc.), among other items.

20.3 Compatibility between Ada 95 and Ada 2005

Although Ada 2005 was designed to be upwards compatible with Ada 95, there are a number of incompatibilities. Several are enumerated below; for a complete description please see the *Annotated Ada 2005 Reference Manual*, or section 9.1.1 in *Rationale for Ada 2005*.

* 'New reserved words.'

  The words `interface`, `overriding` and `synchronized` are reserved in Ada 2005. A pre-Ada 2005 program that uses any of these as an identifier will be illegal.

* 'New declarations in predefined packages.'

  A number of packages in the predefined environment contain new declarations: `Ada.Exceptions`, `Ada.Real_Time`, `Ada.Strings`, `Ada.Strings.Fixed`, `Ada.Strings.Bounded`, `Ada.Strings.Unbounded`, `Ada.Strings.Wide_Fixed`, `Ada.Strings.Wide_Bounded`, `Ada.Strings.Wide_Unbounded`, `Ada.Tags`, `Ada.Text_IO`, and `Interfaces.C`. If an Ada 95 program does a `with` and `use` of any of these packages, the new declarations may cause name clashes.

* 'Access parameters.'

  A nondispatching subprogram with an access parameter cannot be renamed as a dispatching operation. This was permitted in Ada 95.

* 'Access types, discriminants, and constraints.'

  Rule changes in this area have led to some incompatibilities; for example, constrained subtypes of some access types are not permitted in Ada 2005.

* 'Aggregates for limited types.'

  The allowance of aggregates for limited types in Ada 2005 raises the possibility of ambiguities in legal Ada 95 programs, since additional types now need to be considered in expression resolution.

* 'Fixed-point multiplication and division.'

  Certain expressions involving '*' or '/' for a fixed-point type, which were legal in Ada 95 and invoked the predefined versions of these operations, are now ambiguous. The ambiguity may be resolved either by applying a type conversion to the expression, or by explicitly invoking the operation from package `Standard`.

* 'Return-by-reference types.'

  The Ada 95 return-by-reference mechanism has been removed. Instead, the user can declare a function returning a value from an anonymous access type.

20.4 Implementation-dependent characteristics

Although the Ada language defines the semantics of each construct as precisely as practical, in some situations (for example for reasons of efficiency, or where the effect is heavily dependent on the host or target platform) the implementation is allowed some freedom. In porting Ada 83 code to GNAT, you need to be aware of whether / how the existing code exercised such implementation dependencies. Such characteristics fall into several categories, and GNAT offers specific support in assisting the transition from certain Ada 83 compilers.

20.4.1 Implementation-defined pragmas

Ada compilers are allowed to supplement the language-defined pragmas, and these are a potential source of non-portability. All GNAT-defined pragmas are described in [Implementation Defined Pragmas], page 4, and these include several that are specifically intended to correspond to other vendors' Ada 83 pragmas. For migrating from VADS, the pragma `Use_VADS_Size` may be useful. For compatibility with HP Ada 83, GNAT supplies the pragmas `Extend_System`, `Ident`, `Inline_Generic`, `Interface_Name`, `Passive`, `Suppress_All`, and `Volatile`. Other relevant pragmas include `External` and `Link_With`. Some vendor-specific Ada 83 pragmas (`Share_Generic`, `Subtitle`, and `Title`) are recognized, thus avoiding compiler rejection of units that contain such pragmas; they are not relevant in a GNAT context and hence are not otherwise implemented.

20.4.2 Implementation-defined attributes

Analogous to pragmas, the set of attributes may be extended by an implementation. All GNAT-defined attributes are described in [Implementation Defined Attributes], page 120, and these include several that are specifically intended to correspond to other vendors' Ada 83 attributes. For migrating from VADS, the attribute `VADS_Size` may be useful. For compatibility with HP Ada 83, GNAT supplies the attributes `Bit`, `Machine_Size` and `Type_Class`.

20.4.3 Libraries

Vendors may supply libraries to supplement the standard Ada API. If Ada 83 code uses vendor-specific libraries then there are several ways to manage this in Ada 95 and later versions of the standard:

* If the source code for the libraries (specs and bodies) are available, then the libraries can be migrated in the same way as the application.
* If the source code for the specs but not the bodies are available, then you can reimplement the bodies.
* Some features introduced by Ada 95 obviate the need for library support. For example most Ada 83 vendors supplied a package for unsigned integers. The Ada 95 modular type feature is the preferred way to handle this need, so instead of migrating or reimplementing the unsigned integer package it may be preferable to retrofit the application using modular types.

20.4.4 Elaboration order

The implementation can choose any elaboration order consistent with the unit dependency relationship. This freedom means that some orders can result in Program_Error being raised due to an 'Access Before Elaboration': an attempt to invoke a subprogram before its body has been elaborated, or to instantiate a generic before the generic body has been elaborated. By default GNAT attempts to choose a safe order (one that will not encounter access before elaboration problems) by implicitly inserting `Elaborate` or `Elaborate_All` pragmas where needed. However, this can lead to the creation of elaboration circularities and a resulting rejection of the program by gnatbind. This issue is thoroughly described in the 'Elaboration Order Handling in GNAT' appendix in the *GNAT User's Guide*. In brief, there are several ways to deal with this situation:

* Modify the program to eliminate the circularities, e.g., by moving elaboration-time code into explicitly-invoked procedures
* Constrain the elaboration order by including explicit `Elaborate_Body` or `Elaborate` pragmas, and then inhibit the generation of implicit `Elaborate_All` pragmas either globally (as an effect of the '-gnatE' switch) or locally (by selectively suppressing elaboration checks via pragma `Suppress(Elaboration_Check)` when it is safe to do so).

20.4.5 Target-specific aspects

Low-level applications need to deal with machine addresses, data representations, interfacing with assembler code, and similar issues. If such an Ada 83 application is being ported to different target hardware (for example where the byte endianness has changed) then you will need to carefully examine the program logic; the porting effort will heavily depend on the robustness of the original design. Moreover, Ada 95 (and thus Ada 2005, Ada 2012, and Ada 2022) are sometimes incompatible with typical Ada 83 compiler practices regarding implicit packing, the meaning of the Size attribute, and the size of access values. GNAT's approach to these issues is described in [Representation Clauses], page 381.

20.5 Compatibility with Other Ada Systems

If programs avoid the use of implementation dependent and implementation defined features, as documented in the *Ada Reference Manual*, there should be a high degree of portability between GNAT and other Ada systems. The following are specific items which have proved troublesome in moving Ada 95 programs from GNAT to other Ada 95 compilers, but do not affect porting code to GNAT. (As of January 2007, GNAT is the only compiler available for Ada 2005; the following issues may or may not arise for Ada 2005 programs when other compilers appear.)

* 'Ada 83 Pragmas and Attributes'

  Ada 95 compilers are allowed, but not required, to implement the missing Ada 83 pragmas and attributes that are no longer defined in Ada 95. GNAT implements all such pragmas and attributes, eliminating this as a compatibility concern, but some other Ada 95 compilers reject these pragmas and attributes.

* 'Specialized Needs Annexes'

  GNAT implements the full set of special needs annexes. At the current time, it is the only Ada 95 compiler to do so. This means that programs making use of these features may not be portable to other Ada 95 compilation systems.

* 'Representation Clauses'

Some other Ada 95 compilers implement only the minimal set of representation clauses required by the Ada 95 reference manual. GNAT goes far beyond this minimal set, as described in the next section.

20.6 Representation Clauses

The Ada 83 reference manual was quite vague in describing both the minimal required implementation of representation clauses, and also their precise effects. Ada 95 (and thus also Ada 2005) are much more explicit, but the minimal set of capabilities required is still quite limited.

GNAT implements the full required set of capabilities in Ada 95 and Ada 2005, but also goes much further, and in particular an effort has been made to be compatible with existing Ada 83 usage to the greatest extent possible.

A few cases exist in which Ada 83 compiler behavior is incompatible with the requirements in Ada 95 (and thus also Ada 2005). These are instances of intentional or accidental dependence on specific implementation dependent characteristics of these Ada 83 compilers. The following is a list of the cases most likely to arise in existing Ada 83 code.

* 'Implicit Packing'

  Some Ada 83 compilers allowed a Size specification to cause implicit packing of an array or record. This could cause expensive implicit conversions for change of representation in the presence of derived types, and the Ada design intends to avoid this possibility. Subsequent AI's were issued to make it clear that such implicit change of representation in response to a Size clause is inadvisable, and this recommendation is represented explicitly in the Ada 95 (and Ada 2005) Reference Manuals as implementation advice that is followed by GNAT. The problem will show up as an error message rejecting the size clause. The fix is simply to provide the explicit pragma `Pack`, or for more fine tuned control, provide a Component_Size clause.

* 'Meaning of Size Attribute'

  The Size attribute in Ada 95 (and Ada 2005) for discrete types is defined as the minimal number of bits required to hold values of the type. For example, on a 32-bit machine, the size of `Natural` will typically be 31 and not 32 (since no sign bit is required). Some Ada 83 compilers gave 31, and some 32 in this situation. This problem will usually show up as a compile time error, but not always. It is a good idea to check all uses of the 'Size attribute when porting Ada 83 code. The GNAT specific attribute Object_Size can provide a useful way of duplicating the behavior of some Ada 83 compiler systems.

* 'Size of Access Types'

  A common assumption in Ada 83 code is that an access type is in fact a pointer, and that therefore it will be the same size as a System.Address value. This assumption is true for GNAT in most cases with one exception. For the case of a pointer to an unconstrained array type (where the bounds may vary from one value of the access type to another), the default is to use a 'fat pointer', which is represented as two separate pointers, one to the bounds, and one to the array. This representation has a number of advantages, including improved efficiency. However, it may cause some difficulties in porting existing Ada 83 code which makes the assumption that, for example, pointers fit in 32 bits on a machine with 32-bit addressing.

To get around this problem, GNAT also permits the use of 'thin pointers' for access types in this case (where the designated type is an unconstrained array type). These thin pointers are indeed the same size as a System.Address value. To specify a thin pointer, use a size clause for the type, for example:

```
type X is access all String;
for X'Size use Standard'Address_Size;
```

which will cause the type X to be represented using a single pointer. When using this representation, the bounds are right behind the array. This representation is slightly less efficient, and does not allow quite such flexibility in the use of foreign pointers or in using the Unrestricted_Access attribute to create pointers to non-aliased objects. But for any standard portable use of the access type it will work in a functionally correct manner and allow porting of existing code. Note that another way of forcing a thin pointer representation is to use a component size clause for the element size in an array, or a record representation clause for an access field in a record.

See the documentation of Unrestricted_Access in the GNAT RM for a full discussion of possible problems using this attribute in conjunction with thin pointers.

20.7 Compatibility with HP Ada 83

All the HP Ada 83 pragmas and attributes are recognized, although only a subset of them can sensibly be implemented. The description of pragmas in [Implementation Defined Pragmas], page 4, indicates whether or not they are applicable to GNAT.

* 'Default floating-point representation'

  In GNAT, the default floating-point format is IEEE, whereas in HP Ada 83, it is VMS format.

* 'System'

  the package System in GNAT exactly corresponds to the definition in the Ada 95 reference manual, which means that it excludes many of the HP Ada 83 extensions. However, a separate package Aux_DEC is provided that contains the additional definitions, and a special pragma, Extend_System allows this package to be treated transparently as an extension of package System.

21 GNU Free Documentation License

Version 1.3, 3 November 2008

'Preamble'

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

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

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

'1. APPLICABILITY AND DEFINITIONS'

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

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

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

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

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

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

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

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

The "'publisher'" means any person or entity that distributes copies of the Document to the public.

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

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

'2. VERBATIM COPYING'

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute.

However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

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

‘3. COPYING IN QUANTITY’

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

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

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

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

‘4. MODIFICATIONS’

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

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

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

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

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

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

'5. COMBINING DOCUMENTS'

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

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

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

'6. COLLECTIONS OF DOCUMENTS'

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

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

'7. AGGREGATION WITH INDEPENDENT WORKS'

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

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

'8. TRANSLATION'

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

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

‘9. TERMINATION’

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

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

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

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

‘10. FUTURE REVISIONS OF THIS LICENSE’

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

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

‘11. RELICENSING’

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

"Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

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

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

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

'ADDENDUM: How to use this License for your documents'

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

-

-gnat22 option (gcc) 297
-gnatR (gcc) 230

_

___lock file (for shared passive packages) 292

A

Abort_Signal 121
Abstract_State 109
Access 139
Access values 127
Accuracy 179
Accuracy requirements 179
Ada 2005 Language Reference Manual 3
Ada 2022 implementation status 297
Ada 83 attributes 124, 126, 128, 129, 133, 136
Ada 95 Language Reference Manual 3
Ada Extensions 35
Ada.Characters.Handling 170
Ada.Characters.Latin_9 (a-chlat9.ads) 262
Ada.Characters.Wide_Latin_1 (a-cwila1.ads) .. 262
Ada.Characters.Wide_Latin_9 (a-cwila9.ads) .. 262
Ada.Characters.Wide_Wide_Latin_1
(a-chzla1.ads) 262
Ada.Characters.Wide_Wide_Latin_9
(a-chzla9.ads) 263
Ada.Command_Line.Environment
(a-colien.ads) 263
Ada.Command_Line.Remove (a-colire.ads) 263
Ada.Command_Line.Response_File
(a-clrefi.ads) 263
Ada.Containers.Bounded_Holders
(a-coboho.ads) 263
Ada.Direct_IO.C_Streams (a-diocst.ads) 263
Ada.Exceptions.Is_Null_Occurrence
(a-einuoc.ads) 263
Ada.Exceptions.Last_Chance_Handler
(a-elchha.ads) 263
Ada.Exceptions.Traceback (a-exctra.ads) 264
Ada.Sequential_IO.C_Streams (a-siocst.ads) ... 264
Ada.Streams.Stream_IO.C_Streams
(a-ssicst.ads) 264
Ada.Strings.Unbounded.Text_IO
(a-suteio.ads) 264
Ada.Strings.Wide_Unbounded.Wide_Text_IO
(a-swuwti.ads) 264
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Text_IO
(a-szuzti.ads) 264
Ada.Task_Initialization (a-tasini.ads) 264
Ada.Text_IO.C_Streams (a-tiocst.ads) 264
Ada.Text_IO.Reset_Standard_Files
(a-tirsfi.ads) 264
Ada.Wide_Characters.Unicode
(a-wichun.ads) 265
Ada.Wide_Text_IO.C_Streams (a-wtcstr.ads) .. 265
Ada.Wide_Text_IO.Reset_Standard_Files
(a-wrstfi.ads) 265
Ada.Wide_Wide_Characters.Unicode
(a-zchuni.ads) 265
Ada.Wide_Wide_Text_IO.C_Streams
(a-ztcstr.ads) 265
Ada.Wide_Wide_Text_IO.Reset_Standard_Files
(a-zrstfi.ads) 265
Ada_2022 configuration pragma 297
Address 168
Address Clause 222
Address clauses 165
Address image 277
Address of subprogram code 122
Address_Size 121
AI12-0001 (Ada 2022 feature) 297
AI12-0003 (Ada 2022 feature) 297
AI12-0004 (Ada 2022 feature) 297
AI12-0020 (Ada 2022 feature) 297
AI12-0022 (Ada 2022 feature) 298
AI12-0027 (Ada 2022 feature) 298
AI12-0028 (Ada 2022 feature) 298
AI12-0030 (Ada 2022 feature) 298
AI12-0031 (Ada 2022 feature) 298
AI12-0032 (Ada 2022 feature) 298
AI12-0033 (Ada 2022 feature) 298
AI12-0035 (Ada 2022 feature) 298
AI12-0036 (Ada 2022 feature) 299
AI12-0037 (Ada 2022 feature) 299
AI12-0039 (Ada 2022 feature) 299
AI12-0040 (Ada 2022 feature) 299
AI12-0041 (Ada 2022 feature) 299
AI12-0042 (Ada 2022 feature) 299
AI12-0043 (Ada 2022 feature) 299
AI12-0044 (Ada 2022 feature) 300
AI12-0045 (Ada 2022 feature) 300
AI12-0046 (Ada 2022 feature) 300
AI12-0047 (Ada 2022 feature) 300
AI12-0048 (Ada 2022 feature) 300
AI12-0049 (Ada 2022 feature) 300
AI12-0050 (Ada 2022 feature) 301
AI12-0051 (Ada 2022 feature) 301
AI12-0052 (Ada 2022 feature) 301
AI12-0054-2 (Ada 2022 feature) 301
AI12-0055 (Ada 2022 feature) 301
AI12-0059 (Ada 2022 feature) 301
AI12-0061 (Ada 2022 feature) 301
AI12-0062 (Ada 2022 feature) 302
AI12-0065 (Ada 2022 feature) 302
AI12-0067 (Ada 2022 feature) 302

AI12-0068 (Ada 2022 feature) 302
AI12-0069 (Ada 2022 feature) 302
AI12-0070 (Ada 2022 feature) 302
AI12-0071 (Ada 2022 feature) 303
AI12-0072 (Ada 2022 feature) 303
AI12-0073 (Ada 2022 feature) 303
AI12-0074 (Ada 2022 feature) 303
AI12-0075 (Ada 2022 feature) 303
AI12-0076 (Ada 2022 feature) 303
AI12-0077 (Ada 2022 feature) 304
AI12-0078 (Ada 2022 feature) 304
AI12-0081 (Ada 2022 feature) 304
AI12-0084 (Ada 2022 feature) 304
AI12-0085 (Ada 2022 feature) 304
AI12-0086 (Ada 2022 feature) 304
AI12-0088 (Ada 2022 feature) 304
AI12-0089 (Ada 2022 feature) 304
AI12-0093 (Ada 2022 feature) 305
AI12-0094 (Ada 2022 feature) 305
AI12-0095 (Ada 2022 feature) 305
AI12-0096 (Ada 2022 feature) 305
AI12-0097 (Ada 2022 feature) 305
AI12-0098 (Ada 2022 feature) 305
AI12-0099 (Ada 2022 feature) 305
AI12-0100 (Ada 2022 feature) 305
AI12-0101 (Ada 2022 feature) 305
AI12-0102 (Ada 2022 feature) 306
AI12-0103 (Ada 2022 feature) 306
AI12-0104 (Ada 2022 feature) 306
AI12-0105 (Ada 2022 feature) 306
AI12-0106 (Ada 2022 feature) 306
AI12-0107 (Ada 2022 feature) 306
AI12-0109 (Ada 2022 feature) 306
AI12-0110 (Ada 2022 feature) 307
AI12-0112 (Ada 2022 feature) 307
AI12-0114 (Ada 2022 feature) 307
AI12-0116 (Ada 2022 feature) 307
AI12-0117 (Ada 2022 feature) 307
AI12-0120 (Ada 2022 feature) 307
AI12-0121 (Ada 2022 feature) 307
AI12-0124 (Ada 2022 feature) 308
AI12-0125-3 (Ada 2022 feature) 308
AI12-0127 (Ada 2022 feature) 308
AI12-0128 (Ada 2022 feature) 308
AI12-0129 (Ada 2022 feature) 308
AI12-0130 (Ada 2022 feature) 308
AI12-0131 (Ada 2022 feature) 308
AI12-0132 (Ada 2022 feature) 308
AI12-0133 (Ada 2022 feature) 309
AI12-0135 (Ada 2022 feature) 309
AI12-0136 (Ada 2022 feature) 309
AI12-0137 (Ada 2022 feature) 309
AI12-0138 (Ada 2022 feature) 309
AI12-0140 (Ada 2022 feature) 309
AI12-0143 (Ada 2022 feature) 309
AI12-0144 (Ada 2022 feature) 309
AI12-0145 (Ada 2022 feature) 310
AI12-0147 (Ada 2022 feature) 310
AI12-0149 (Ada 2022 feature) 310
AI12-0150 (Ada 2022 feature) 310
AI12-0154 (Ada 2022 feature) 310
AI12-0156 (Ada 2022 feature) 310
AI12-0157 (Ada 2022 feature) 310
AI12-0160 (Ada 2022 feature) 311
AI12-0162 (Ada 2022 feature) 311
AI12-0164 (Ada 2022 feature) 311
AI12-0165 (Ada 2022 feature) 311
AI12-0166 (Ada 2022 feature) 311
AI12-0167 (Ada 2022 feature) 311
AI12-0168 (Ada 2022 feature) 311
AI12-0169 (Ada 2022 feature) 311
AI12-0170 (Ada 2022 feature) 311
AI12-0172 (Ada 2022 feature) 312
AI12-0173 (Ada 2022 feature) 312
AI12-0174 (Ada 2022 feature) 312
AI12-0175 (Ada 2022 feature) 312
AI12-0179 (Ada 2022 feature) 312
AI12-0180 (Ada 2022 feature) 312
AI12-0181 (Ada 2022 feature) 312
AI12-0182 (Ada 2022 feature) 312
AI12-0184 (Ada 2022 feature) 312
AI12-0185 (Ada 2022 feature) 313
AI12-0186 (Ada 2022 feature) 313
AI12-0187 (Ada 2022 feature) 313
AI12-0191 (Ada 2022 feature) 313
AI12-0192 (Ada 2022 feature) 313
AI12-0194 (Ada 2022 feature) 313
AI12-0195 (Ada 2022 feature) 313
AI12-0196 (Ada 2022 feature) 314
AI12-0198 (Ada 2022 feature) 314
AI12-0199 (Ada 2022 feature) 314
AI12-0201 (Ada 2022 feature) 314
AI12-0203 (Ada 2022 feature) 314
AI12-0204 (Ada 2022 feature) 314
AI12-0205 (Ada 2022 feature) 314
AI12-0206 (Ada 2022 feature) 314
AI12-0207 (Ada 2022 feature) 315
AI12-0208 (Ada 2022 feature) 315
AI12-0211 (Ada 2022 feature) 315
AI12-0212 (Ada 2022 feature) 315
AI12-0216 (Ada 2022 feature) 315
AI12-0217 (Ada 2022 feature) 315
AI12-0220 (Ada 2022 feature) 315
AI12-0222 (Ada 2022 feature) 316
AI12-0225 (Ada 2022 feature) 316
AI12-0226 (Ada 2022 feature) 316
AI12-0227 (Ada 2022 feature) 316
AI12-0228 (Ada 2022 feature) 316
AI12-0231 (Ada 2022 feature) 316
AI12-0232 (Ada 2022 feature) 316
AI12-0233 (Ada 2022 feature) 316
AI12-0234 (Ada 2022 feature) 317
AI12-0235 (Ada 2022 feature) 317
AI12-0236 (Ada 2022 feature) 317
AI12-0237 (Ada 2022 feature) 317
AI12-0242 (Ada 2022 feature) 317

AI12-0247 (Ada 2022 feature) 317
AI12-0249 (Ada 2022 feature) 317
AI12-0250 (Ada 2022 feature) 317
AI12-0252 (Ada 2022 feature) 318
AI12-0254 (Ada 2022 feature) 318
AI12-0256 (Ada 2022 feature) 318
AI12-0258 (Ada 2022 feature) 318
AI12-0259 (Ada 2022 feature) 318
AI12-0260 (Ada 2022 feature) 318
AI12-0261 (Ada 2022 feature) 318
AI12-0262 (Ada 2022 feature) 319
AI12-0263 (Ada 2022 feature) 319
AI12-0264 (Ada 2022 feature) 319
AI12-0265 (Ada 2022 feature) 319
AI12-0269 (Ada 2022 feature) 319
AI12-0272 (Ada 2022 feature) 319
AI12-0275 (Ada 2022 feature) 319
AI12-0277 (Ada 2022 feature) 319
AI12-0278 (Ada 2022 feature) 320
AI12-0279 (Ada 2022 feature) 320
AI12-0280-2 (Ada 2022 feature) 320
AI12-0282 (Ada 2022 feature) 320
AI12-0285 (Ada 2022 feature) 320
AI12-0287 (Ada 2022 feature) 321
AI12-0289 (Ada 2022 feature) 321
AI12-0290 (Ada 2022 feature) 321
AI12-0291 (Ada 2022 feature) 321
AI12-0293 (Ada 2022 feature) 321
AI12-0295 (Ada 2022 feature) 321
AI12-0301 (Ada 2022 feature) 321
AI12-0304 (Ada 2022 feature) 322
AI12-0306 (Ada 2022 feature) 322
AI12-0307 (Ada 2022 feature) 322
AI12-0309 (Ada 2022 feature) 322
AI12-0311 (Ada 2022 feature) 322
AI12-0315 (Ada 2022 feature) 322
AI12-0318 (Ada 2022 feature) 322
AI12-0321 (Ada 2022 feature) 322
AI12-0325 (Ada 2022 feature) 322
AI12-0329 (Ada 2022 feature) 323
AI12-0331 (Ada 2022 feature) 323
AI12-0333 (Ada 2022 feature) 323
AI12-0335 (Ada 2022 feature) 323
AI12-0336 (Ada 2022 feature) 323
AI12-0337 (Ada 2022 feature) 323
AI12-0338 (Ada 2022 feature) 323
AI12-0339 (Ada 2022 feature) 323
AI12-0340 (Ada 2022 feature) 324
AI12-0342 (Ada 2022 feature) 324
AI12-0343 (Ada 2022 feature) 324
AI12-0345 (Ada 2022 feature) 324
AI12-0350 (Ada 2022 feature) 324
AI12-0351 (Ada 2022 feature) 324
AI12-0352 (Ada 2022 feature) 324
AI12-0356 (Ada 2022 feature) 324
AI12-0363 (Ada 2022 feature) 325
AI12-0364 (Ada 2022 feature) 325
AI12-0366 (Ada 2022 feature) 325
AI12-0367 (Ada 2022 feature) 325
AI12-0368 (Ada 2022 feature) 325
AI12-0369 (Ada 2022 feature) 325
AI12-0372 (Ada 2022 feature) 325
AI12-0373 (Ada 2022 feature) 325
AI12-0376 (Ada 2022 feature) 326
AI12-0377 (Ada 2022 feature) 326
AI12-0381 (Ada 2022 feature) 326
AI12-0382 (Ada 2022 feature) 326
AI12-0383 (Ada 2022 feature) 326
AI12-0384-2 (Ada 2022 feature) 326
AI12-0385 (Ada 2022 feature) 326
AI12-0389 (Ada 2022 feature) 326
AI12-0394 (Ada 2022 feature) 327
AI12-0395 (Ada 2022 feature) 327
AI12-0397 (Ada 2022 feature) 327
AI12-0398 (Ada 2022 feature) 327
AI12-0399 (Ada 2022 feature) 327
AI12-0400 (Ada 2022 feature) 327
AI12-0401 (Ada 2022 feature) 328
AI12-0402 (Ada 2022 feature) 328
AI12-0409 (Ada 2022 feature) 328
AI12-0411 (Ada 2022 feature) 328
AI12-0412 (Ada 2022 feature) 328
AI12-0413 (Ada 2022 feature) 328
AI12-0423 (Ada 2022 feature) 329
AI12-0432 (Ada 2022 feature) 329
Alignment 63, 129, 137, 203
Alignment Clause 202
Alignment clauses 165
Alignments of components 20
allocator 137
Alternative Character Sets 161
AltiVec 265, 266
Always_Terminates 109
Annex E 291
Annotate 109
Anonymous access types 228
Argument passing mechanisms 31
argument removal 263
Array packing 40
Array splitter 266
Arrays 163, 269, 276
as private type 168
Asm_Input 121
Asm_Output 121
Assert_Failure 278
Assertions 15, 17, 278
Async_Readers 110
Async_Writers 110
Atomic Synchronization 27, 31
Atomic_Always_Lock_Free 122
Attribute 223
Attribute Loop_Entry 98
Attribute Old 98
AWK 266

B

Biased representation 207
Big endian 123
Binary search 266
Bind environment 266
Bit 122
Bit ordering 168
bit ordering 211
Bit_Order Clause 211
Bit_Position 122
Boolean_Entry_Barriers 156
Bounded Buffers 266
Bounded errors 160
Bounded-length strings 170
Branch Prediction 266
Bubble sort 267
byte ordering 212
Byte swapping 267

C

C 172
C streams 277
C Streams 263, 264, 265
Calendar 267
casing 36
Casing of External names 36
Casing utilities 267
CGI (Common Gateway Interface) 268
CGI (Common Gateway Interface) cookie support 268
CGI (Common Gateway Interface) debugging 268
Character handling (‘‘GNAT.Case_Util‘‘) 267
Character Sets 161
Check names 16
Check pragma control 17
Checks 67, 70, 72, 163
Child Units 160
COBOL 173
COBOL support 177
Code_Address 122
Command line 263, 268
Compatibility (between Ada 83 and Ada 95 / Ada 2005 / Ada 2012 / Ada 2022) 375
Compatibility between Ada 95 and Ada 2005 378
Compilation_Date 199
Compilation_ISO_Date 199
Compilation_Time 199
Compiler Version 268
Compiler_Version 123
complex arithmetic 179
Complex arithmetic accuracy 179
Complex elementary functions 178
Complex types 177
Component Clause 219
Component_Size (in pragma Component_Alignment) 20
Component_Size Clause 210
Component_Size clauses 166
Component_Size_4 (in pragma Component_Alignment) 20
configuration pragma Ada_2022 297
Constant_After_Elaboration 110
Constrained 123
Containers 171
Contract cases 21
Contract_Cases 110
control 17
Controlling assertions 17
Convention 227
Convention for anonymous access types 228
Conventions 3, 22
Conversion 279
Cookie support in CGI 268
CRC32 267
Current exception 268
Current time 276
Cyclic Redundancy Check 267

D

Debug pools 268
Debugging 268, 270
debugging with Initialize_Scalars 45
Dec Ada 83 casing compatibility 36
DEC Ada 83 34
Decimal radix support 177
Decoding strings 269
Decoding UTF-8 strings 269
default 203
Default (in pragma Component_Alignment) 20
default settings 63
Default_Bit_Order 123
Default_Initial_Condition 110
Default_Scalar_Storage_Order 25, 123
Default_Storage_Pool 26
Deferring aborts 5
defining 16
Defining check names 16
Depends 110
Deref 123
Descriptor 123
Descriptor_Size 123
determination of 230
Dimension 110
Dimension_System 111
Directory operations 269
Directory operations iteration 269
Disable_Controlled 112
Discriminants 127
Distribution Systems Annex 291
Dope vector 123
Dump Memory 272

Duration'Small 163

E

effect on representation 227
Effective_Reads 112
Effective_Writes 112
Elab_Body 124
Elab_Spec 124
Elab_Subp_Body 124
Elaborated 124
Elaboration control 28
Elimination of unused subprograms 28
Emax 124
Enabled 125
Enclosing_Entity 200
Encoding strings 269
Encoding UTF-8 strings 269
Endianness 133, 267
Entry queuing policies 176
Enum_Rep 125
Enum_Val 125
enumeration 167
Enumeration representation clauses 167
Enumeration values 162
Environment entries 263
Epsilon 126
Error detection 160
exception 72, 278
Exception 272
Exception actions 270
Exception information 163
Exception retrieval 268
Exception traces 270
Exception_Information' 200
Exception_Message 72, 200
Exception_Name 200
Exceptional_Cases 31, 112
exceptions 270
Exceptions 270
Exit_Cases 31, 112
Export 171, 224
extendable 269, 276
extending 34
extensions for unbounded strings 264
extensions for unbounded wide strings 264
extensions for unbounded wide wide strings ... 264
Extensions_Visible 113
External Names 36

F

Fast_Math 126
Favor_Top_Level 114
File 200
File locking 271
Finalization_Size 126
Fixed_Value 126
Float types 162
Floating-point overflow 16
Floating-Point Processor 270
foreign 276
Foreign threads 276
Forking a new process 290
Formal container for vectors 263
Formatted String 270
Fortran 173
From_Address 126
From_Any 127

G

Get_Immediate 170, 248, 277
Ghost 114
Ghost_Predicate 114
global 278
Global 114
Global storage pool 278
GNAT Extensions 35
GNAT.Altivec (g-altive.ads) 265
GNAT.Altivec.Conversions (g-altcon.ads) 265
GNAT.Altivec.Vector_Operations
(g-alveop.ads) 266
GNAT.Altivec.Vector_Types (g-alvety.ads) 266
GNAT.Altivec.Vector_Views (g-alvevi.ads) 266
GNAT.Array_Split (g-arrspl.ads) 266
GNAT.AWK (g-awk.ads) 266
GNAT.Binary_Search (g-binsea.ads) 266
GNAT.Bind_Environment (g-binenv.ads) 266
GNAT.Bounded_Buffers (g-boubuf.ads) 266
GNAT.Bounded_Mailboxes (g-boumai.ads) 266
GNAT.Branch_Prediction (g-brapre.ads) 266
GNAT.Bubble_Sort (g-bubsor.ads) 267
GNAT.Bubble_Sort_A (g-busora.ads) 267
GNAT.Bubble_Sort_G (g-busorg.ads) 267
GNAT.Byte_Order_Mark (g-byorma.ads) 267
GNAT.Byte_Swapping (g-bytswa.ads) 267
GNAT.C_Time (g-c_time.ads) 267
GNAT.Calendar (g-calend.ads) 267
GNAT.Calendar.Time_IO (g-catiio.ads) 267
GNAT.Case_Util (g-casuti.ads) 267
GNAT.CGI (g-cgi.ads) 268
GNAT.CGI.Cookie (g-cgicoo.ads) 268
GNAT.CGI.Debug (g-cgideb.ads) 268
GNAT.Command_Line (g-comlin.ads) 268
GNAT.Compiler_Version (g-comver.ads) 268
GNAT.CRC32 (g-crc32.ads) 267
GNAT.Ctrl_C (g-ctrl_c.ads) 268
GNAT.Current_Exception (g-curexc.ads) 268

GNAT.Debug_Pools (g-debpoo.ads) 268
GNAT.Debug_Utilities (g-debuti.ads) 268
GNAT.Decode_String (g-decstr.ads) 269
GNAT.Decode_UTF8_String (g-deutst.ads) 269
GNAT.Directory_Operations (g-dirope.ads) 269
GNAT.Directory_Operations.Iteration (g-diopit.ads) 269
GNAT.Dynamic_HTables (g-dynhta.ads) 269
GNAT.Dynamic_Tables (g-dyntab.ads) 269
GNAT.Encode_String (g-encstr.ads) 269
GNAT.Encode_UTF8_String (g-enutst.ads) 269
GNAT.Exception_Actions (g-excact.ads) 270
GNAT.Exception_Traces (g-exctra.ads) 270
GNAT.Exceptions (g-except.ads) 270
GNAT.Expect (g-expect.ads) 270
GNAT.Expect.TTY (g-exptty.ads) 270
GNAT.Float_Control (g-flocon.ads) 270
GNAT.Formatted_String (g-forstr.ads) 270
GNAT.Generic_Fast_Math_Functions (g-gfmafu.ads) 270
GNAT.Heap_Sort (g-heasor.ads) 271
GNAT.Heap_Sort_A (g-hesora.ads) 271
GNAT.Heap_Sort_G (g-hesorg.ads) 271
GNAT.HTable (g-htable.ads) 271
GNAT.IO (g-io.ads) 271
GNAT.IO_Aux (g-io_aux.ads) 271
GNAT.Lock_Files (g-locfil.ads) 271
GNAT.MBBS_Discrete_Random (g-mbdira.ads) 272
GNAT.MBBS_Float_Random (g-mbflra.ads) 272
GNAT.MD5 (g-md5.ads) 272
GNAT.Memory_Dump (g-memdum.ads) 272
GNAT.Most_Recent_Exception (g-moreex.ads) 272
GNAT.OS_Lib (g-os_lib.ads) 272
GNAT.Perfect_Hash_Generators (g-pehage.ads) 272
GNAT.Random_Numbers (g-rannum.ads) 272
GNAT.Regexp (g-regexp.ads) 273
GNAT.Registry (g-regist.ads) 273
GNAT.Regpat (g-regpat.ads) 273
GNAT.Rewrite_Data (g-rewdat.ads) 273
GNAT.Secondary_Stack_Info (g-sestin.ads) 273
GNAT.Semaphores (g-semaph.ads) 273
GNAT.Serial_Communications (g-sercom.ads) 273
GNAT.SHA1 (g-sha1.ads) 273
GNAT.SHA224 (g-sha224.ads) 273
GNAT.SHA256 (g-sha256.ads) 274
GNAT.SHA384 (g-sha384.ads) 274
GNAT.SHA512 (g-sha512.ads) 274
GNAT.Signals (g-signal.ads) 274
GNAT.Sockets (g-socket.ads) 274
GNAT.Source_Info (g-souinf.ads) 274
GNAT.Spelling_Checker (g-speche.ads) 274
GNAT.Spelling_Checker_Generic (g-spchge.ads) 274
GNAT.Spitbol (g-spitbo.ads) 275
GNAT.Spitbol.Patterns (g-spipat.ads) 274
GNAT.Spitbol.Table_Boolean (g-sptabo.ads) 275
GNAT.Spitbol.Table_Integer (g-sptain.ads) 275
GNAT.Spitbol.Table_VString (g-sptavs.ads) 275
GNAT.SSE (g-sse.ads) 275
GNAT.SSE.Vector_Types (g-ssvety.ads) 275
GNAT.String_Hash (g-strhas.ads) 275
GNAT.String_Split (g-strspl.ads) 275
GNAT.Strings (g-string.ads) 275
GNAT.Table (g-table.ads) 276
GNAT.Task_Lock (g-tasloc.ads) 276
GNAT.Threads (g-thread.ads) 276
GNAT.Time_Stamp (g-timsta.ads) 276
GNAT.Traceback (g-traceb.ads) 276
GNAT.Traceback.Symbolic (g-trasym.ads) 276
GNAT.UTF_32 (g-utf_32.ads) 276
GNAT.UTF_32_Spelling_Checker (g-u3spch.ads) 276
GNAT.Wide_Spelling_Checker (g-wispch.ads) 277
GNAT.Wide_String_Split (g-wistsp.ads) 277
GNAT.Wide_Wide_Spelling_Checker (g-zspche.ads) 277
GNAT.Wide_Wide_String_Split (g-zistsp.ads) 277

H

handling long command lines 263
Handling of Records with Holes 220
Has_Access_Values 127
Has_Discriminants 127
Has_Tagged_Values 127
Hash functions 272, 275
Hash tables 269, 271
Heap usage 169

I

I/O interfacing 277
IBM Packed Format 277
Image 277
Img 127
Immediate_Reclamation 145
Implementation-dependent features 2
implicit 169
Import 224
Initial_Condition 114
Initialization 94
Initialized 128
Initializes 114
Inline_Always 114
Input/Output facilities 271
Integer maps 275
Integer types 162
Integer_Value 128
Interfaces 171
Interfaces.C.Extensions (i-cexten.ads) 277

Interfaces.C.Streams (i-cstrea.ads) 277
Interfaces.Packed_Decimal (i-pacdec.ads) 277
Interfaces.VxWorks (i-vxwork.ads) 277
Interfaces.VxWorks.IO (i-vxwoio.ads) 277
interfacing 277
Interfacing to C++ 24, 76
Interfacing to VxWorks 277
Interfacing to VxWorks' I/O 277
interfacing with 172, 173
Interfacing with "Text_IO" 264
Interfacing with "Wide_Text_IO" 265
Interfacing with "Wide_Wide_Text_IO" 265
Interfacing with C++ 23, 24
Interfacing with Direct_IO 263
Interfacing with Sequential_IO 264
Interfacing with Stream_IO 264
Interrupt 268
Interrupt support 174
Interrupts 175
Intrinsic operator 199
Intrinsic Subprograms 199
Invalid representations 14
Invalid values 14
Invalid_Value 128
Invariant 114
Invariant'Class 114
IO support 264
Iterable 114

L

Large 128
Latin-1 377
Latin_1 constants for Wide_Character 262
Latin_1 constants for Wide_Wide_Character 262
Latin_9 constants for Character 262
Latin_9 constants for Wide_Character 262
Latin_9 constants for Wide_Wide_Character 263
Library_Level 128
License checking 50
Line 200
Linker_Section 115
Little endian 123
local 279
Local storage pool 279
Local_Restrictions 115
Lock_Free 116
Locking 276
Locking Policies 176
Locking using files 271
Loop_Entry 129

M

Machine Code insertions 287
Machine operations 173
Machine_Size 129
Mailboxes 266
Mantissa 129
Maps 275
Mathematical functions 270
Max_Asynchronous_Select_Nesting 145
Max_Entry_Queue_Depth 145
Max_Entry_Queue_Length 145
Max_Integer_Size 129
Max_Protected_Entries 145
Max_Queue_Length 116
Max_Select_Alternatives 145
Max_Storage_At_Blocking 146
Max_Task_Entries 146
Max_Tasks 146
maximum 129
Maximum_Alignment 129
Maximum_Alignment attribute 202
Mechanism_Code 130
Memory allocation 278
Memory corruption debugging 268
Memory-mapped I/O 227
Message Digest MD5 272
monotonic 176
multidimensional 163
Multidimensional arrays 163
Multiprocessor interface 278

N

Named assertions 15, 17
Named numbers 139
No_Abort_Statements 146
No_Access_Parameter_Allocators 146
No_Access_Subprograms 146
No_Allocators 146
No_Anonymous_Allocators 146
No_Asynchronous_Control 146
No_Caching 116
No_Calendar 146
No_Coextensions 146
No_Default_Initialization 147
No_Delay 147
No_Dependence 147
No_Direct_Boolean_Operators 147
No_Dispatch 147
No_Dispatching_Calls 147
No_Dynamic_Accessibility_Checks 157
No_Dynamic_Attachment 149
No_Dynamic_Interrupts 149
No_Dynamic_Priorities 149
No_Dynamic_Sized_Objects 157
No_Elaboration_Code 156
No_Elaboration_Code_All 116
No_Entry_Calls_In_Elaboration_Code 149

No_Entry_Queue 158
No_Enumeration_Maps 149
No_Exception_Handlers 149
No_Exception_Propagation 149
No_Exception_Registration 150
No_Exceptions 150
No_Finalization 150
No_Fixed_Point 150
No_Floating_Point 150
No_Implementation_Aspect_Specifications 158
No_Implementation_Attributes 158
No_Implementation_Identifiers 158
No_Implementation_Pragmas 158
No_Implementation_Restrictions 158
No_Implementation_Units 158
No_Implicit_Aliasing 158
No_Implicit_Conditionals 150
No_Implicit_Dynamic_Code 151
No_Implicit_Heap_Allocations 151
No_Implicit_Loops 159
No_Implicit_Protected_Object_Allocations 151
No_Implicit_Task_Allocations 151
No_Initialize_Scalars 151
No_Inline 116
No_IO 151
No_Local_Allocators 151
No_Local_Protected_Objects 151
No_Local_Tagged_Types 151
No_Local_Timing_Events 152
No_Long_Long_Integers 152
No_Multiple_Elaboration 152
No_Nested_Finalization 152
No_Obsolescent_Features 159
No_Protected_Type_Allocators 152
No_Protected_Types 152
No_Raise 116
No_Recursion 152
No_Reentrancy 152
No_Relative_Delay 152
No_Requeue 152
No_Requeue_Statements 152
No_Secondary_Stack 153
No_Select_Statements 153
No_Specific_Termination_Handlers 153
No_Specification_of_Aspect 153
No_Standard_Allocators_After_Elaboration 153
No_Standard_Storage_Pools 153
No_Stream_Optimizations 153
No_Streams 153
No_Tagged_Streams 117
No_Tagged_Type_Registration 154
No_Task_Allocators 154
No_Task_At_Interrupt_Priority 154
No_Task_Attributes 154
No_Task_Attributes_Package 154
No_Task_Hierarchy 154
No_Task_Parts 117
No_Task_Termination 154
No_Tasking 154
No_Terminate_Alternatives 154
No_Unchecked_Access 155
No_Unchecked_Conversion 155
No_Unchecked_Deallocation 155
No_Use_Of_Attribute 155
No_Use_Of_Entity 155
No_Use_Of_Pragma 155
No_Wide_Characters 159
Null_Occurrence 263
Null_Parameter 130
Numerics 177

O

Object_Size 117, 130, 207
Obsolescent 117
obtaining most recent 272
of an address 277
of bits 211
of bytes 212
of compiler 268
of objects 207
Old 131
on "Address" 168
Operating System interface 272
Operations 168
operations of 168
ordering 211, 212
Overlaying of objects 224

P

Package "Interrupts" 175
Package Interfaces 171
Package Task_Attributes 175
Packed Decimal 277
Packed types 164
Parameters 130, 131
Parsing 266
Part_Of 117
Partition communication subsystem 177
Partition interfacing functions 278
Passed_By_Reference 131
passing 130
Passing by copy 15
passing mechanism 130
Pattern matching 273, 274
PCS 177
Persistent_BSS 117
Pool_Address 131
Portability 2
Post 67, 70
Postcondition 67
postconditions 67, 70
Potentially_Invalid 117
Pragma 202
pragma Ada_2022 297

Pragma Component_Alignment 20
Pragma Pack (for arrays) 216
Pragma Pack (for records) 218
Pragma Pack (for type Natural)............... 217
Pragma Pack warning......................... 217
pragma Shared_Passive 291
Pragmas 80, 160
Pre 70
Pre-elaboration requirements.................. 175
Pre_Class 72
Preconditions 70
preconditions............................... 70, 72
Predicate 117
Preemptive abort 176
Prefix_Exception_Messages..................... 72
Program_Exit................................ 117
Protected procedure handlers 175
Pure... 270
Pure packages 270
Pure_Barriers................................ 155
Pure_Function 117

R

Random number generation 170, 272
Range_Length 132
Rational compatibility 65
Rational profile 65, 102
Rational Profile................................ 40
Read attribute............................... 170
Real-Time Systems Annex compliance......... 290
Record Representation Clause................. 219
Record representation clauses 167
records 167
Refined_Depends 118
Refined_Global 118
Refined_Initialization 118
Refined_Post................................. 118
Refined_State................................ 118
Regular expressions 273
Remote_Access_Type 118
Removing command line arguments........... 263
representation 202
Representation........................... 230, 279
Representation Clause 202
Representation Clauses 202
Representation clauses 164, 167
representation of............................. 139
Representation of enums 125
Representation of wide characters 279
Representation Pragma 202
response file 263
Response file for command line................ 263
Restriction_Set 132
Restrictions.................................. 132
Restrictions definitions....................... 279
Result 133
Return values................................ 130

Rewrite data 273
Rotate_Left.................................. 201
Rotate_Right 201
Round....................................... 133
Run-time restrictions access.................. 279

S

Safe_Emax................................... 133
Safe_Large................................... 133
Safe_Small................................... 133
Scalar storage order.......................... 133
Scalar_Storage_Order 25, 118, 133
Secondary Stack Info 273
Secondary_Stack_Size......................... 118
Secure Hash Algorithm SHA-1 273
Secure Hash Algorithm SHA-224.............. 273
Secure Hash Algorithm SHA-256.............. 274
Secure Hash Algorithm SHA-384.............. 274
Secure Hash Algorithm SHA-512.............. 274
Semaphores.................................. 273
Sequential elaboration policy.................. 179
Serial_Communications 273
Sets of strings 275
setting for not-first subtype 144
Shared....................................... 118
Shared passive packages....................... 291
SHARED_MEMORY_DIRECTORY
environment variable 291
Shift operators................................ 76
Shift_Left.................................... 201
Shift_Right 201
Shift_Right_Arithmetic 201
Side_Effects.................................. 118
Signals 274
simple 84, 136
Simple I/O 271
Simple storage pool 84, 136
Simple_Barriers.............................. 156
Simple_Storage_Pool 118, 136
Simple_Storage_Pool_Type.................... 118
size.. 205
Size 102, 130, 144, 205, 207
Size Clause 203
Size clauses.................................. 166
Size for biased representation 207
Size of “Address“............................ 121
Small................................... 136, 137
Small_Denominator 137
Small_Numerator 137
Sockets...................................... 274
Sorting 267, 271
Source Information 274
Source_Location 201
SPARK_05 159
SPARK_Mode................................ 119
Spawn capability 272
Spell checking 274, 276, 277

SPITBOL interface 275
SPITBOL pattern matching 274
SPITBOL Tables 275
Static_Dispatch_Tables 159
Static_Priorities 156
Static_Storage_Size 156
Storage place attributes 168
Storage pool 84, 136, 278, 279
Storage_Size Clause 204
Storage_Unit 137
Storage_Unit (in pragma Component_Alignment) 20
Stream files 248
Stream operations 279
Stream oriented attributes 169, 170
String decoding 269
String encoding 269
String maps 275
String splitter 275
String stream operations 279
Stub_Type 137
Subprogram address 122
Subprogram_Variant 91, 119
subtypes 203
Super 347
Suppress_Debug_Info 119
Suppress_Initialization 119
Suppressing external name 32, 33, 34
Suppressing initialization 94
suppression of 94, 163
Suppression of checks 163
synonyms 22, 80
System 34
System.Address_Image (s-addima.ads) 277
System.Assertions (s-assert.ads) 278
System.Atomic_Counters (s-atocou.ads) 278
System.Memory (s-memory.ads) 278
System.Multiprocessors (s-multip.ads) 278
System.Multiprocessors.Dispatching_Domains (s-mudido.ads) 278
System.Partition_Interface (s-parint.ads) 278
System.Pool_Global (s-pooglo.ads) 278
System.Pool_Local (s-pooloc.ads) 279
System.Restrictions (s-restri.ads) 279
System.Rident (s-rident.ads) 279
System.Strings.Stream_Ops (s-ststop.ads) 279
System.Unsigned_Types (s-unstyp.ads) 279
System.Wch_Cnv (s-wchcnv.ads) 279
System.Wch_Con (s-wchcon.ads) 279
System_Allocator_Alignment 137

T

Table implementation 269, 276
Tagged values 127
Target_Name 137
Task locking 276
Task specific storage 96
Task synchronization 276
Task_Attributes 96, 175
Tasking restrictions 176
Test cases 95
Test_Case 119
testing for 127, 263
Text_IO 264, 271
Text_IO extensions 248
Text_IO for unbounded strings 248
Text_IO operations 248
Text_IO resetting standard files 264
Thread_Local_Storage 119
Threads 276
Time 176, 267
Time stamp 276
TLS (Thread Local Storage) 96
To_Address 138, 223
To_Any 138
Trace back facilities 276
Traceback for Exception Occurrence 264
trampoline 151
Type_Class 138
Type_Key 138
TypeCode 138
typographical 3
Typographical conventions 3

U

Unbounded_String 248, 264
Unbounded_Wide_String 264
Unbounded_Wide_Wide_String 264
Unchecked conversion 168
Unchecked deallocation 169
Unconstrained_Array 139
Unevaluated_Use_Of_Old 98
Unicode 269
Unicode categorization 265
Unions in C 98
Universal_Aliasing 119
Universal_Literal_String 139
unmodified 99
Unmodified 119
Unreferenced 119
unreferenced 100, 101
Unreferenced_Objects 119
unrestricted 139
Unrestricted_Access 139
unused 102
Update 142
used for objects 130
User_Aspect 119

UTF-8 269
UTF-8 representation 267
UTF-8 string decoding 269
UTF-8 string encoding 269

V

VADS compatibility 102, 144
VADS_Size 144
Valid_Scalars 143
Valid_Value 143
Value_Size 120, 144, 207
variant record objects 205
Variant record objects 205
Version 268
Volatile_Full_Access 120
Volatile_Function 120
VxWorks 277

W

Warnings 99, 100, 101, 102, 120
Wchar_T_Size 144
when passed by reference 131
Wide characte representations 267
Wide Character 279
Wide character codes 276
Wide character decoding 269
Wide character encoding 269
Wide String 279
Wide_Character 265
Wide_String splitter 277
Wide_Text_IO resetting standard files 265
Wide_Wide_Character 265
Wide_Wide_String splitter 277
Wide_Wide_Text_IO resetting standard files 265
Windows Registry 273
Word_Size 144
Write attribute 170

X

XDR representation 170

Z

Zero address 130