

Learning Ada

The complete contents
of learn.adacore.com

LEARN.
ADACORE.COM



Learning Ada

Release 2023-01

Various authors

Jan 28, 2023

CONTENTS

I Introduction to Ada	3
1 Introduction	7
1.1 History	7
1.2 Ada today	7
1.3 Philosophy	8
1.4 SPARK	8
2 Imperative language	9
2.1 Hello world	9
2.2 Imperative language - If/Then/Else	10
2.3 Imperative language - Loops	12
2.3.1 For loops	13
2.3.2 Bare loops	14
2.3.3 While loops	15
2.4 Imperative language - Case statement	16
2.5 Imperative language - Declarative regions	17
2.6 Imperative language - conditional expressions	19
2.6.1 If expressions	19
2.6.2 Case expressions	20
3 Subprograms	21
3.1 Subprograms	21
3.1.1 Subprogram calls	22
3.1.2 Nested subprograms	23
3.1.3 Function calls	24
3.2 Parameter modes	25
3.3 Subprogram calls	26
3.3.1 In parameters	26
3.3.2 In out parameters	26
3.3.3 Out parameters	27
3.3.4 Forward declaration of subprograms	28
3.4 Renaming	29
4 Modular programming	31
4.1 Packages	31
4.2 Using a package	33
4.3 Package body	33
4.4 Child packages	35
4.4.1 Child of a child package	36
4.4.2 Multiple children	37
4.4.3 Visibility	38
4.5 Renaming	40
5 Strongly typed language	43
5.1 What is a type?	43

5.2	Integers	43
5.2.1	Operational semantics	45
5.3	Unsigned types	46
5.4	Enumerations	47
5.5	Floating-point types	48
5.5.1	Basic properties	48
5.5.2	Precision of floating-point types	49
5.5.3	Range of floating-point types	50
5.6	Strong typing	51
5.7	Derived types	54
5.8	Subtypes	55
5.8.1	Subtypes as type aliases	57
6	Records	61
6.1	Record type declaration	61
6.2	Aggregates	62
6.3	Component selection	62
6.4	Renaming	63
7	Arrays	67
7.1	Array type declaration	67
7.2	Indexing	70
7.3	Simpler array declarations	71
7.4	Range attribute	72
7.5	Unconstrained arrays	73
7.6	Predefined array type: String	74
7.7	Restrictions	76
7.8	Returning unconstrained arrays	77
7.9	Declaring arrays (2)	78
7.10	Array slices	79
7.11	Renaming	80
8	More about types	83
8.1	Aggregates: A primer	83
8.2	Overloading and qualified expressions	84
8.3	Character types	86
9	Access types (pointers)	89
9.1	Overview	89
9.2	Allocation (by type)	91
9.3	Dereferencing	92
9.4	Other features	92
9.5	Mutually recursive types	93
10	More about records	95
10.1	Dynamically sized record types	95
10.2	Records with discriminant	96
10.3	Variant records	98
11	Fixed-point types	101
11.1	Decimal fixed-point types	101
11.2	Ordinary fixed-point types	103
12	Privacy	107
12.1	Basic encapsulation	107
12.2	Abstract data types	108
12.3	Limited types	110
12.4	Child packages & privacy	111

13 Generics	115
13.1 Introduction	115
13.2 Formal type declaration	115
13.3 Formal object declaration	116
13.4 Generic body definition	116
13.5 Generic instantiation	117
13.6 Generic packages	118
13.7 Formal subprograms	119
13.8 Example: I/O instances	121
13.9 Example: ADTs	123
13.10 Example: Swap	124
13.11 Example: Reversing	127
13.12 Example: Test application	130
14 Exceptions	135
14.1 Exception declaration	135
14.2 Raising an exception	135
14.3 Handling an exception	136
14.4 Predefined exceptions	138
15 Tasking	139
15.1 Tasks	139
15.1.1 Simple task	139
15.1.2 Simple synchronization	140
15.1.3 Delay	142
15.1.4 Synchronization: rendezvous	143
15.1.5 Select loop	144
15.1.6 Cycling tasks	145
15.2 Protected objects	149
15.2.1 Simple object	149
15.2.2 Entries	150
15.3 Task and protected types	151
15.3.1 Task types	152
15.3.2 Protected types	153
16 Design by contracts	155
16.1 Pre- and postconditions	155
16.2 Predicates	158
16.3 Type invariants	162
17 Interfacing with C	165
17.1 Multi-language project	165
17.2 Type convention	165
17.3 Foreign subprograms	166
17.3.1 Calling C subprograms in Ada	166
17.3.2 Calling Ada subprograms in C	167
17.4 Foreign variables	168
17.4.1 Using C global variables in Ada	168
17.4.2 Using Ada variables in C	170
17.5 Generating bindings	171
17.5.1 Adapting bindings	172
18 Object-oriented programming	177
18.1 Derived types	178
18.2 Tagged types	179
18.3 Classwide types	180
18.4 Dispatching operations	182
18.5 Dot notation	183
18.6 Private & Limited	184

18.7 Classwide access types	186
19 Standard library: Containers	191
19.1 Vectors	191
19.1.1 Instantiation	191
19.1.2 Initialization	192
19.1.3 Appending and prepending elements	193
19.1.4 Accessing first and last elements	194
19.1.5 Iterating	195
19.1.6 Finding and changing elements	200
19.1.7 Inserting elements	201
19.1.8 Removing elements	202
19.1.9 Other Operations	205
19.2 Sets	208
19.2.1 Initialization and iteration	208
19.2.2 Operations on elements	209
19.2.3 Other Operations	211
19.3 Indefinite maps	213
19.3.1 Hashed maps	214
19.3.2 Ordered maps	215
19.3.3 Complexity	216
20 Standard library: Dates & Times	217
20.1 Date and time handling	217
20.1.1 Delaying using date	218
20.2 Real-time	221
20.2.1 Benchmarking	221
21 Standard library: Strings	225
21.1 String operations	225
21.2 Limitation of fixed-length strings	229
21.3 Bounded strings	230
21.4 Unbounded strings	232
22 Standard library: Files and streams	235
22.1 Text I/O	235
22.2 Sequential I/O	237
22.3 Direct I/O	239
22.4 Stream I/O	241
23 Standard library: Numerics	245
23.1 Elementary Functions	245
23.2 Random Number Generation	246
23.3 Complex Types	248
23.4 Vector and Matrix Manipulation	250
24 Appendices	255
24.1 Appendix A: Generic Formal Types	255
24.1.1 Indefinite version	257
24.2 Appendix B: Containers	257
II Introduction To SPARK	259
25 SPARK Overview	263
25.1 What is it?	263
25.2 What do the tools do?	264
25.3 Key Tools	264
25.4 A trivial example	264

25.5 The Programming Language	265
25.6 Limitations	265
25.6.1 No side-effects in expressions	265
25.6.2 No aliasing of names	267
25.7 Designating SPARK Code	269
25.8 Code Examples / Pitfalls	270
25.8.1 Example #1	270
25.8.2 Example #2	271
25.8.3 Example #3	272
25.8.4 Example #4	273
25.8.5 Example #5	274
25.8.6 Example #6	275
25.8.7 Example #7	276
25.8.8 Example #8	276
25.8.9 Example #9	277
25.8.10 Example #10	278
26 Flow Analysis	281
26.1 What does flow analysis do?	281
26.2 Errors Detected	281
26.2.1 Uninitialized Variables	281
26.2.2 Ineffective Statements	282
26.2.3 Incorrect Parameter Mode	284
26.3 Additional Verifications	285
26.3.1 Global Contracts	285
26.3.2 Depends Contracts	286
26.4 Shortcomings	288
26.4.1 Modularity	288
26.4.2 Composite Types	289
26.4.3 Value Dependency	291
26.4.4 Contract Computation	292
26.5 Code Examples / Pitfalls	293
26.5.1 Example #1	293
26.5.2 Example #2	294
26.5.3 Example #3	295
26.5.4 Example #4	296
26.5.5 Example #5	297
26.5.6 Example #6	298
26.5.7 Example #7	299
26.5.8 Example #8	301
26.5.9 Example #9	302
26.5.10 Example #10	303
27 Proof of Program Integrity	305
27.1 Runtime Errors	305
27.2 Modularity	307
27.2.1 Exceptions	308
27.3 Contracts	310
27.3.1 Executable Semantics	311
27.3.2 Additional Assertions and Contracts	313
27.4 Debugging Failed Proof Attempts	314
27.4.1 Debugging Errors in Code or Specification	314
27.4.2 Debugging Cases where more Information is Required	316
27.4.3 Debugging Prover Limitations	317
27.5 Code Examples / Pitfalls	319
27.5.1 Example #1	320
27.5.2 Example #2	321
27.5.3 Example #3	322

27.5.4 Example #4	323
27.5.5 Example #5	324
27.5.6 Example #6	325
27.5.7 Example #7	326
27.5.8 Example #8	327
27.5.9 Example #9	328
27.5.10 Example #10	329
28 State Abstraction	331
28.1 What's an Abstraction	331
28.2 Why is Abstraction Useful?	332
28.3 Abstraction of a Package's State	333
28.4 Declaring a State Abstraction	333
28.5 Refining an Abstract State	334
28.6 Representing Private Variables	335
28.7 Additional State	336
28.7.1 Nested Packages	336
28.7.2 Constants that Depend on Variables	337
28.8 Subprogram Contracts	338
28.8.1 Global and Depends	338
28.8.2 Preconditions and Postconditions	341
28.9 Initialization of Local Variables	343
28.10 Code Examples / Pitfalls	345
28.10.1 Example #1	345
28.10.2 Example #2	346
28.10.3 Example #3	347
28.10.4 Example #4	348
28.10.5 Example #5	349
28.10.6 Example #6	350
28.10.7 Example #7	351
28.10.8 Example #8	352
28.10.9 Example #9	354
28.10.10 Example #10	355
29 Proof of Functional Correctness	357
29.1 Beyond Program Integrity	357
29.2 Advanced Contracts	360
29.2.1 Ghost Code	361
29.2.2 Ghost Functions	363
29.2.3 Global Ghost Variables	365
29.3 Guide Proof	367
29.3.1 Local Ghost Variables	367
29.3.2 Ghost Procedures	369
29.3.3 Handling of Loops	370
29.3.4 Loop Invariants	371
29.4 Code Examples / Pitfalls	377
29.4.1 Example #1	377
29.4.2 Example #2	378
29.4.3 Example #3	380
29.4.4 Example #4	381
29.4.5 Example #5	382
29.4.6 Example #6	384
29.4.7 Example #7	385
29.4.8 Example #8	386
29.4.9 Example #9	387
29.4.10 Example #10	388

III Introduction to Embedded Systems Programming	391
30 Introduction	395
30.1 So, what will we actually cover?	395
30.2 Definitions	396
30.3 Down To The Bare Metal	396
30.4 The Ada Drivers Library	397
31 Low Level Programming	399
31.1 Separation Principle	399
31.2 Guaranteed Level of Support	400
31.3 Querying Implementation Limits and Characteristics	401
31.4 Querying Representation Choices	404
31.5 Specifying Representation	409
31.6 Unchecked Programming	424
31.7 Data Validity	433
32 Multi-Language Development	435
32.1 General Interfacing	436
32.1.1 Aspect/Pragma Convention	436
32.1.2 Aspect/Pragma Import and Export	439
32.1.3 Aspect/Pragma External_Name and Link_Name	440
32.1.4 Package Interfaces	441
32.2 Language-Specific Interfacing	443
32.2.1 Package Interfaces.C	443
32.2.2 Package Interfaces.C.Strings	448
32.2.3 Package Interfaces.C.Pointers	449
32.2.4 Package Interfaces.Fortran	449
32.2.5 Machine Code Insertions (MCI)	450
32.3 When Ada Is Not the Main Language	454
33 Interacting with Devices	457
33.1 Non-Memory-Mapped Devices	459
33.2 Memory-Mapped Devices	460
33.3 Dynamic Address Conversion	467
33.4 Address Arithmetic	470
34 General-Purpose Code Generators	473
34.1 Aspect Independent	474
34.2 Aspect Volatile	476
34.3 Aspect Atomic	479
34.4 Aspect Full_Access_Only	480
35 Handling Interrupts	485
35.1 Background	485
35.2 Language-Defined Interrupt Model	489
35.3 Interrupt Handlers	490
35.4 Interrupt Management	493
35.5 Associating Handlers With Interrupts	494
35.6 Interrupt Priorities	496
35.7 Common Design Idioms	499
35.7.1 Parameterizing Handlers	499
35.7.2 Multi-Level Handlers	501
35.8 Final Points	507
36 Conclusion	509

IV What's New in Ada 2022	511
37 Introduction	515
37.1 References	515
38 'Image attribute for any type	517
38.1 'Image attribute for a value	517
38.2 'Image attribute for any type	517
38.3 References	518
39 Redefining the 'Image attribute	519
39.1 What's the Root_Buffer_Type?	520
39.2 Outdated draft implementation	520
39.3 References	520
40 User-Defined Literals	521
40.1 Turn Ada into JavaScript	522
40.2 References	523
41 Advanced Array Aggregates	525
41.1 Square brackets	525
41.2 Iterated Component Association	526
41.3 References	527
42 Container Aggregates	529
42.1 References	533
43 Delta Aggregates	535
43.1 Delta aggregate for records	535
43.2 Delta aggregate for arrays	535
43.3 References	536
44 Target Name Symbol (@)	537
44.1 Alternatives	539
44.2 References	539
45 Enumeration representation	541
45.1 Literal positions	541
45.2 Representation values	542
45.3 Before Ada 2022	543
45.4 References	544
46 Big Numbers	545
46.1 Big Integers	545
46.2 Tiny RSA implementation	545
46.3 Big Reals	547
46.4 References	547
47 Interfacing C variadic functions	549
47.1 References	550
V Ada for the C++ or Java Developer	551
48 Preface	555
49 Basics	557
50 Compilation Unit Structure	559

51 Statements, Declarations, and Control Structures	561
51.1 Statements and Declarations	561
51.2 Conditions	563
51.3 Loops	564
52 Type System	567
52.1 Strong Typing	567
52.2 Language-Defined Types	568
52.3 Application-Defined Types	568
52.4 Type Ranges	570
52.5 Generalized Type Contracts: Subtype Predicates	571
52.6 Attributes	571
52.7 Arrays and Strings	572
52.8 Heterogeneous Data Structures	575
52.9 Pointers	576
53 Functions and Procedures	581
53.1 General Form	581
53.2 Overloading	583
53.3 Subprogram Contracts	583
54 Packages	585
54.1 Declaration Protection	585
54.2 Hierarchical Packages	586
54.3 Using Entities from Packages	586
55 Classes and Object Oriented Programming	589
55.1 Primitive Subprograms	589
55.2 Derivation and Dynamic Dispatch	590
55.3 Constructors and Destructors	593
55.4 Encapsulation	594
55.5 Abstract Types and Interfaces	594
55.6 Invariants	596
56 Generics	599
56.1 Generic Subprograms	599
56.2 Generic Packages	600
56.3 Generic Parameters	601
57 Exceptions	603
57.1 Standard Exceptions	603
57.2 Custom Exceptions	604
58 Concurrency	605
58.1 Tasks	605
58.2 Rendezvous	608
58.3 Selective Rendezvous	610
58.4 Protected Objects	611
59 Low Level Programming	615
59.1 Representation Clauses	615
59.2 Embedded Assembly Code	616
59.3 Interfacing with C	617
60 Conclusion	619
61 References	621

VI Ada for the Embedded C Developer	623
62 Introduction	627
62.1 So, what is this Ada thing anyway?	627
62.2 Ada — The Technical Details	629
63 The C Developer's Perspective on Ada	631
63.1 What we mean by Embedded Software	631
63.2 The GNAT Toolchain	631
63.3 The GNAT Toolchain for Embedded Targets	632
63.4 Hello World in Ada	633
63.5 The Ada Syntax	634
63.6 Compilation Unit Structure	635
63.7 Packages	635
63.7.1 Declaration Protection	635
63.7.2 Hierarchical Packages	636
63.7.3 Using Entities from Packages	637
63.8 Statements and Declarations	637
63.9 Conditions	643
63.10 Loops	646
63.11 Type System	652
63.11.1 Strong Typing	652
63.11.2 Language-Defined Types	655
63.11.3 Application-Defined Types	655
63.11.4 Type Ranges	658
63.11.5 Unsigned And Modular Types	661
63.11.6 Attributes	664
63.11.7 Arrays and Strings	666
63.11.8 Heterogeneous Data Structures	673
63.11.9 Pointers	674
63.12 Functions and Procedures	679
63.12.1 General Form	679
63.12.2 Overloading	682
63.12.3 Aspects	684
64 Concurrency and Real-Time	687
64.1 Understanding the various options	687
64.2 Tasks	687
64.3 Rendezvous	690
64.4 Selective Rendezvous	691
64.5 Protected Objects	694
64.6 Ravenscar	697
65 Writing Ada on Embedded Systems	701
65.1 Understanding the Ada Run-Time	701
65.2 Low Level Programming	702
65.2.1 Representation Clauses	702
65.2.2 Embedded Assembly Code	703
65.3 Interrupt Handling	704
65.4 Dealing with Absence of FPU with Fixed Point	706
65.5 Volatile and Atomic data	710
65.5.1 Volatile	710
65.5.2 Atomic	712
65.6 Interfacing with Devices	713
65.6.1 Size aspect and attribute	714
65.6.2 Register overlays	715
65.6.3 Data streams	718
65.7 ARM and svd2ada	723

66 Enhancing Verification with SPARK and Ada	725
66.1 Understanding Exceptions and Dynamic Checks	725
66.2 Understanding Dynamic Checks versus Formal Proof	731
66.3 Initialization and Correct Data Flow	734
66.4 Contract-Based Programming	736
66.5 Replacing Defensive Code	738
66.6 Proving Absence of Run-Time Errors	740
66.7 Proving Abstract Properties	741
66.8 Final Comments	742
67 C to Ada Translation Patterns	743
67.1 Naming conventions and casing considerations	743
67.2 Manually interfacing C and Ada	743
67.3 Building and Debugging mixed language code	745
67.4 Automatic interfacing	746
67.5 Using Arrays in C interfaces	746
67.6 By-value vs. by-reference types	748
67.7 Naming and prefixes	749
67.8 Pointers	750
67.9 Bitwise Operations	752
67.10 Mapping Structures to Bit-Fields	754
67.10.1 Overlays vs. Unchecked Conversions	767
68 Handling Variability and Re-usability	771
68.1 Understanding static and dynamic variability	771
68.2 Handling variability & reusability statically	771
68.2.1 Genericity	771
68.2.2 Simple derivation	774
68.2.3 Configuration pragma files	779
68.2.4 Configuration packages	781
68.3 Handling variability & reusability dynamically	785
68.3.1 Records with discriminants	785
68.3.2 Variant records	787
68.3.3 Object orientation	794
68.3.4 Pointer to subprograms	807
68.4 Design by components using dynamic libraries	812
69 Performance considerations	815
69.1 Overall expectations	815
69.2 Switches and optimizations	815
69.2.1 Optimizations levels	815
69.2.2 Inlining	816
69.3 Checks and assertions	817
69.3.1 Checks	817
69.3.2 Assertions	821
69.4 Dynamic vs. static structures	821
69.5 Pointers vs. data copies	822
69.5.1 Function returns	825
70 Argumentation and Business Perspectives	829
70.1 What's the expected ROI of a C to Ada transition?	829
70.2 Who is using Ada today?	830
70.3 What is the future of the Ada technology?	830
70.4 Is the Ada toolset complete?	831
70.5 Where can I find Ada or SPARK developers?	831
70.6 How to introduce Ada and SPARK in an existing code base?	832
71 Conclusion	833

72 Appendix A: Hands-On Object-Oriented Programming	837
72.1 System Overview	837
72.2 Non Object-Oriented Approach	838
72.2.1 Starting point in C	838
72.2.2 Initial translation to Ada	841
72.2.3 Improved Ada implementation	845
72.3 First Object-Oriented Approach	849
72.3.1 Interfaces	849
72.3.2 Base type	849
72.3.3 Derived types	850
72.3.4 Subprograms from parent	851
72.3.5 Type AB	852
72.3.6 Updated source-code	852
72.4 Further Improvements	856
72.4.1 Dispatching calls	856
72.4.2 Dynamic allocation	858
72.4.3 Limited controlled types	859
72.4.4 Updated source-code	859
VII SPARK Ada for the MISRA C Developer	865
73 Preface	869
74 Enforcing Basic Program Consistency	871
74.1 Taming Text-Based Inclusion	871
74.2 Hardening Link-Time Checking	874
74.3 Going Towards Encapsulation	875
75 Enforcing Basic Syntactic Guarantees	879
75.1 Distinguishing Code and Comments	879
75.2 Specially Handling Function Parameters and Result	880
75.2.1 Handling the Result of Function Calls	880
75.2.2 Handling Function Parameters	881
75.3 Ensuring Control Structures Are Not Abused	882
75.3.1 Preventing the Semicolon Mistake	882
75.3.2 Avoiding Complex Switch Statements	883
75.3.3 Avoiding Complex Loops	885
75.3.4 Avoiding the Dangling Else Issue	887
76 Enforcing Strong Typing	889
76.1 Enforcing Strong Typing for Pointers	889
76.1.1 Pointers Are Not Addresses	890
76.1.2 Pointers Are Not References	890
76.1.3 Pointers Are Not Arrays	891
76.1.4 Pointers Should Be Typed	894
76.2 Enforcing Strong Typing for Scalars	896
76.2.1 Restricting Operations on Types	897
76.2.2 Restricting Explicit Conversions	901
76.2.3 Restricting Implicit Conversions	901
77 Initializing Data Before Use	905
77.1 Detecting Reads of Uninitialized Data	905
77.2 Detecting Partial or Redundant Initialization of Arrays and Structures	910
78 Controlling Side Effects	913
78.1 Preventing Undefined Behavior	913
78.2 Reducing Programmer Confusion	914
78.3 Side Effects and SPARK	915

79 Detecting Undefined Behavior	919
79.1 Preventing Undefined Behavior in SPARK	919
79.2 Proof of Absence of Run-Time Errors in SPARK	920
80 Detecting Unreachable Code and Dead Code	925
81 Conclusion	929
82 References	931
82.1 About MISRA C	931
82.2 About SPARK	932
82.3 About MISRA C and SPARK	932
VIII Introduction to GNAT Toolchain	933
83 GNAT Community	937
83.1 Installation	937
83.2 Basic commands	938
83.3 Compiler warnings	938
83.3.1 -gnatwa switch and warning suppression	938
83.3.2 Style checking	941
84 GPRbuild	943
84.1 Basic commands	943
84.2 Project files	943
84.2.1 Basic structure	943
84.2.2 Customization	944
84.3 Project dependencies	945
84.3.1 Simple dependency	945
84.3.2 Dependencies to dynamic libraries	947
84.4 Configuration pragma files	947
84.5 Configuration packages	948
85 GNAT Studio	951
85.1 Start-up	951
85.1.1 Windows	951
85.1.2 Linux	951
85.2 Creating projects	951
85.3 Building	952
85.4 Debugging	952
85.4.1 Debug information	952
85.4.2 Improving main application	953
85.4.3 Debugging the application	954
85.5 Formal verification	954
86 GNAT Tools	957
86.1 gnatchop	957
86.2 gnatprep	958
86.3 gnatmem	960
86.4 gnatmetric	961
86.5 gnatdoc	961
86.6 gnatpp	963
86.7 gnatstub	964
IX Introduction to Ada: Laboratories	965
87 Imperative language	969
87.1 Hello World	969

87.2 Greetings	969
87.3 Positive Or Negative	970
87.4 Numbers	971
88 Subprograms	973
88.1 Subtract procedure	973
88.2 Subtract function	974
88.3 Equality function	975
88.4 States	977
88.5 States #2	978
88.6 States #3	979
88.7 States #4	980
89 Modular Programming	983
89.1 Months	983
89.2 Operations	984
90 Strongly typed language	987
90.1 Colors	987
90.2 Integers	989
90.3 Temperatures	993
91 Records	997
91.1 Directions	997
91.2 Colors	999
91.3 Inventory	1003
92 Arrays	1007
92.1 Constrained Array	1007
92.2 Colors: Lookup-Table	1009
92.3 Unconstrained Array	1012
92.4 Product info	1015
92.5 String_10	1018
92.6 List of Names	1020
93 More About Types	1025
93.1 Aggregate Initialization	1025
93.2 Versioning	1027
93.3 Simple todo list	1029
93.4 Price list	1031
94 Privacy	1037
94.1 Directions	1037
94.2 Limited Strings	1039
94.3 Bonus exercise	1043
94.3.1 Colors	1044
94.3.2 List of Names	1044
94.3.3 Price List	1044
95 Generics	1045
95.1 Display Array	1045
95.2 Average of Array of Float	1047
95.3 Average of Array of Any Type	1049
95.4 Generic list	1052
96 Exceptions	1055
96.1 Uninitialized Value	1055
96.2 Numerical Exception	1057
96.3 Re-raising Exceptions	1059

97 Tasking	1063
97.1 Display Service	1063
97.2 Event Manager	1064
97.3 Generic Protected Queue	1066
98 Design by contracts	1069
98.1 Price Range	1069
98.2 Pythagorean Theorem: Predicate	1070
98.3 Pythagorean Theorem: Precondition	1072
98.4 Pythagorean Theorem: Postcondition	1074
98.5 Pythagorean Theorem: Type Invariant	1076
98.6 Primary Color	1078
99 Object-oriented programming	1083
99.1 Simple type extension	1083
99.2 Online Store	1085
10 Standard library: Containers	1091
100.1 Simple todo list	1091
100.2 List of unique integers	1093
10 Standard library: Dates & Times	1097
101.1 Holocene calendar	1097
101.2 List of events	1098
10 Standard library: Strings	1103
102.1 Concatenation	1103
102.2 List of events	1105
10 Standard library: Numerics	1109
103.1 Decibel Factor	1109
103.2 Root-Mean-Square	1111
103.3 Rotation	1114
10 Solutions	1119
104.1 Imperative Language	1119
104.1.1 Hello World	1119
104.1.2 Greetings	1119
104.1.3 Positive Or Negative	1120
104.1.4 Numbers	1120
104.2 Subprograms	1121
104.2.1 Subtract Procedure	1121
104.2.2 Subtract Function	1122
104.2.3 Equality function	1123
104.2.4 States	1124
104.2.5 States #2	1125
104.2.6 States #3	1126
104.2.7 States #4	1127
104.3 Modular Programming	1128
104.3.1 Months	1128
104.3.2 Operations	1129
104.4 Strongly typed language	1131
104.4.1 Colors	1131
104.4.2 Integers	1133
104.4.3 Temperatures	1136
104.5 Records	1138
104.5.1 Directions	1138
104.5.2 Colors	1140
104.5.3 Inventory	1143

104.6	Arrays	1145
104.6.1	Constrained Array	1145
104.6.2	Colors: Lookup-Table	1147
104.6.3	Unconstrained Array	1149
104.6.4	Product info	1151
104.6.5	String_10	1153
104.6.6	List of Names	1155
104.7	More About Types	1158
104.7.1	Aggregate Initialization	1158
104.7.2	Versioning	1160
104.7.3	Simple todo list	1161
104.7.4	Price list	1163
104.8	Privacy	1165
104.8.1	Directions	1165
104.8.2	Limited Strings	1167
104.9	Generics	1170
104.9.1	Display Array	1170
104.9.2	Average of Array of Float	1172
104.9.3	Average of Array of Any Type	1173
104.9.4	Generic list	1175
104.10	Exceptions	1177
104.10.1	Uninitialized Value	1177
104.10.2	Numerical Exception	1179
104.10.3	Re-raising Exceptions	1181
104.11	Tasking	1182
104.11.1	Display Service	1182
104.11.2	Event Manager	1184
104.11.3	Generic Protected Queue	1185
104.12	Design by contracts	1188
104.12.1	Price Range	1188
104.12.2	Pythagorean Theorem: Predicate	1189
104.12.3	Pythagorean Theorem: Precondition	1191
104.12.4	Pythagorean Theorem: Postcondition	1192
104.12.5	Pythagorean Theorem: Type Invariant	1194
104.12.6	Primary Colors	1196
104.13	Object-oriented programming	1198
104.13.1	Simple type extension	1198
104.13.2	Online Store	1200
104.14	Standard library: Containers	1203
104.14.1	Simple todo list	1203
104.14.2	List of unique integers	1204
104.15	Standard library: Dates & Times	1206
104.15.1	Holocene calendar	1206
104.15.2	List of events	1207
104.16	Standard library: Strings	1209
104.16.1	Concatenation	1209
104.16.2	List of events	1211
104.17	Standard library: Numerics	1214
104.17.1	Decibel Factor	1214
104.17.2	Root-Mean-Square	1215
104.17.3	Rotation	1217

X Bug Free Coding with SPARK Ada

1221

105	Let's Build a Stack	1225
105.1	Background	1225
105.2	Input Format	1228

105.3 Constraints	1228
105.4 Output Format	1228
105.5 Sample Input	1228
105.6 Sample Output	1228

Bibliography

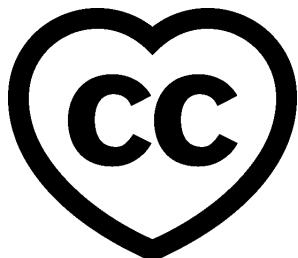
1233

Part I

Introduction to Ada

Copyright © 2018 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page¹](#)



This course will teach you the basics of the Ada programming language and is intended for those who already have a basic understanding of programming techniques. You will learn how to apply those techniques to programming in Ada.

This document was written by Raphaël Amiard and Gustavo A. Hoffmann, with review from Richard Kenner.

¹ <http://creativecommons.org/licenses/by-sa/4.0>

INTRODUCTION

1.1 History

In the 1970s the United States Department of Defense (DOD) suffered from an explosion of the number of programming languages, with different projects using different and non-standard dialects or language subsets / supersets. The DOD decided to solve this problem by issuing a request for proposals for a common, modern programming language. The winning proposal was one submitted by Jean Ichbiah from CII Honeywell-Bull.

The first Ada standard was issued in 1983; it was subsequently revised and enhanced in 1995, 2005 and 2012, with each revision bringing useful new features.

This tutorial will focus on Ada 2012 as a whole, rather than teaching different versions of the language.

1.2 Ada today

Today, Ada is heavily used in embedded real-time systems, many of which are safety critical. While Ada is and can be used as a general-purpose language, it will really shine in low-level applications:

- Embedded systems with low memory requirements (no garbage collector allowed).
- Direct interfacing with hardware.
- Soft or hard real-time systems.
- Low-level systems programming.

Specific domains seeing Ada usage include Aerospace & Defense, civil aviation, rail, and many others. These applications require a high degree of safety: a software defect is not just an annoyance, but may have severe consequences. Ada provides safety features that detect defects at an early stage — usually at compilation time or using static analysis tools. Ada can also be used to create applications in a variety of other areas, such as:

- Video game programming²
- Real-time audio³
- Kernel modules⁴

This is a non-comprehensive list that hopefully sheds light on which kind of programming Ada is good at.

² <https://github.com/AdaDoom3/AdaDoom3>

³ <http://www.electronicdesign.com/embedded-revolution/assessing-ada-language-audio-applications>

⁴ <http://www.nihamkin.com/tag/kernel.html>

In terms of modern languages, the closest in terms of targets and level of abstraction are probably C++⁵ and Rust⁶.

1.3 Philosophy

Ada's philosophy is different from most other languages. Underlying Ada's design are principles that include the following:

- Readability is more important than conciseness. Syntactically this shows through the fact that keywords are preferred to symbols, that no keyword is an abbreviation, etc.
- Very strong typing. It is very easy to introduce new types in Ada, with the benefit of preventing data usage errors.
 - It is similar to many functional languages in that regard, except that the programmer has to be much more explicit about typing in Ada, because there is almost no type inference.
- Explicit is better than implicit. Although this is a Python⁷ commandment, Ada takes it way further than any language we know of:
 - There is mostly no structural typing, and most types need to be explicitly named by the programmer.
 - As previously said, there is mostly no type inference.
 - Semantics are very well defined, and undefined behavior is limited to an absolute minimum.
 - The programmer can generally give a *lot* of information about what their program means to the compiler (and other programmers). This allows the compiler to be extremely helpful (read: strict) with the programmer.

During this course, we will explain the individual language features that are building blocks for that philosophy.

1.4 SPARK

While this class is solely about the Ada language, it is worth mentioning that another language, extremely close to and interoperable with Ada, exists: the SPARK language.

SPARK is a subset of Ada, designed so that the code written in SPARK is amenable to automatic proof. This provides a level of assurance with regard to the correctness of your code that is much higher than with a regular programming language.

There is a dedicated [course for the SPARK language](#) (page 261) but keep in mind that every time we speak about the specification power of Ada during this course, it is power that you can leverage in SPARK to help proving the correctness of program properties ranging from absence of run-time errors to compliance with formally specified functional requirements.

⁵ <https://en.wikipedia.org/wiki/C%2B%2B>

⁶ <https://www.rust-lang.org/en-US/>

⁷ <https://www.python.org>

IMPERATIVE LANGUAGE

Ada is a multi-paradigm language with support for object orientation and some elements of functional programming, but its core is a simple, coherent procedural/imperative language akin to C or Pascal.

In other languages

One important distinction between Ada and a language like C is that statements and expressions are very clearly distinguished. In Ada, if you try to use an expression where a statement is required then your program will fail to compile. This rule supports a useful stylistic principle: expressions are intended to deliver values, not to have side effects. It can also prevent some programming errors, such as mistakenly using the equality operator = instead of the assignment operation := in an assignment statement.

2.1 Hello world

Here's a very simple imperative Ada program:

Listing 1: greet.adb

```
1 with Ada.Text_Io;  
2  
3 procedure Greet is  
4 begin  
5     -- Print "Hello, World!" to the screen  
6     Ada.Text_Io.Put_Line ("Hello, World!");  
7 end Greet;
```

Runtime output

```
Hello, World!
```

which we'll assume is in the source file `greet.adb`.

There are several noteworthy things in the above program:

- A subprogram in Ada can be either a procedure or a function. A procedure, as illustrated above, does not return a value when called.
- **with** is used to reference external modules that are needed in the procedure. This is similar to `import` in various languages or roughly similar to `#include` in C and C++. We'll see later how they work in detail. Here, we are requesting a standard library module, the `Ada.Text_Io` package, which contains a procedure to print text on the screen: `Put_Line`.

- Greet is a procedure, and the main entry point for our first program. Unlike in C or C++, it can be named anything you prefer. The builder will determine the entry point. In our simple example, **gprbuild**, GNAT's builder, will use the file you passed as parameter.
- Put_Line is a procedure, just like Greet, except it is declared in the Ada.Text_I0 module. It is the Ada equivalent of C's printf.
- Comments start with `--` and go to the end of the line. There is no multi-line comment syntax, that is, it is not possible to start a comment in one line and continue it in the next line. The only way to create multiple lines of comments in Ada is by using `--` on each line. For example:

```
-- We start a comment in this line...
-- and we continue on the second line...
```

In other languages

Procedures are similar to functions in C or C++ that return **void**. We'll see later how to declare functions in Ada.

Here is a minor variant of the "Hello, World" example:

Listing 2: greet.adb

```
1 with Ada.Text_I0; use Ada.Text_I0;
2
3 procedure Greet is
4 begin
5     -- Print "Hello, World!" to the screen
6     Put_Line ("Hello, World!");
7 end Greet;
```

Runtime output

```
Hello, World!
```

This version utilizes an Ada feature known as a **use** clause, which has the form **use package-name**. As illustrated by the call on Put_Line, the effect is that entities from the named package can be referenced directly, without the *package-name*. prefix.

2.2 Imperative language - If/Then/Else

This section describes Ada's **if** statement and introduces several other fundamental language facilities including integer I/O, data declarations, and subprogram parameter modes.

Ada's **if** statement is pretty unsurprising in form and function:

Listing 3: check_positive.adb

```
1 with Ada.Text_I0; use Ada.Text_I0;
2 with Ada.Integer_Text_I0; use Ada.Integer_Text_I0;
3
4 procedure Check_Positive is
5     N : Integer;
6 begin
7     -- Put a String
8     Put ("Enter an integer value: ");
```

(continues on next page)

(continued from previous page)

```

9      -- Read in an integer value
10     Get (N);
11
12     if N > 0 then
13         -- Put an Integer
14         Put (N);
15         Put_Line (" is a positive number");
16     end if;
17 end Check_Positive;

```

The **if** statement minimally consists of the reserved word **if**, a condition (which must be a Boolean value), the reserved word **then** and a non-empty sequence of statements (the **then** part) which is executed if the condition evaluates to True, and a terminating **end if**.

This example declares an integer variable **N**, prompts the user for an integer, checks if the value is positive and, if so, displays the integer's value followed by the string " is a positive number". If the value is not positive, the procedure does not display any output.

The type **Integer** is a predefined signed type, and its range depends on the computer architecture. On typical current processors **Integer** is 32-bit signed.

The example illustrates some of the basic functionality for integer input-output. The relevant subprograms are in the predefined package **Ada.Integer_Text_IO** and include the **Get** procedure (which reads in a number from the keyboard) and the **Put** procedure (which displays an integer value).

Here's a slight variation on the example, which illustrates an **if** statement with an **else** part:

Listing 4: check_positive.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4  procedure Check_Positive is
5      N : Integer;
6  begin
7      -- Put a String
8      Put ("Enter an integer value: ");
9
10     -- Reads in an integer value
11     Get (N);
12
13     -- Put an Integer
14     Put (N);
15
16     if N > 0 then
17         Put_Line (" is a positive number");
18     else
19         Put_Line (" is not a positive number");
20     end if;
21 end Check_Positive;

```

In this example, if the input value is not positive then the program displays the value followed by the String " is not a positive number".

Our final variation illustrates an **if** statement with **elsif** sections:

Listing 5: check_direction.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Check_Direction is
5      N : Integer;
6  begin
7      Put ("Enter an integer value: ");
8      Get (N);
9      Put (N);
10
11     if N = 0 or N = 360 then
12         Put_Line (" is due north");
13     elsif N in 1 .. 89 then
14         Put_Line (" is in the northeast quadrant");
15     elsif N = 90 then
16         Put_Line (" is due east");
17     elsif N in 91 .. 179 then
18         Put_Line (" is in the southeast quadrant");
19     elsif N = 180 then
20         Put_Line (" is due south");
21     elsif N in 181 .. 269 then
22         Put_Line (" is in the southwest quadrant");
23     elsif N = 270 then
24         Put_Line (" is due west");
25     elsif N in 271 .. 359 then
26         Put_Line (" is in the northwest quadrant");
27     else
28         Put_Line (" is not in the range 0..360");
29     end if;
30 end Check_Direction;
```

This example expects the user to input an integer between 0 and 360 inclusive, and displays which quadrant or axis the value corresponds to. The `in` operator in Ada tests whether a scalar value is within a specified range and returns a Boolean result. The effect of the program should be self-explanatory; later we'll see an alternative and more efficient style to accomplish the same effect, through a `case` statement.

Ada's `elsif` keyword differs from C or C++, where nested `else .. if` blocks would be used instead. And another difference is the presence of the `end if` in Ada, which avoids the problem known as the "dangling else".

2.3 Imperative language - Loops

Ada has three ways of specifying loops. They differ from the C / Java / Javascript for-loop, however, with simpler syntax and semantics in line with Ada's philosophy.

2.3.1 For loops

The first kind of loop is the **for** loop, which allows iteration through a discrete range.

Listing 6: greet_5a.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet_5a is
4    begin
5      for I in 1 .. 5 loop
6          -- Put_Line is a procedure call
7          Put_Line ("Hello, World!" & Integer'Image (I));
8          -- ^ Procedure parameter
9      end loop;
10 end Greet_5a;
```

Runtime output

```
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

A few things to note:

- **1 .. 5** is a discrete range, from **1** to **5** inclusive.
- The loop parameter **I** (the name is arbitrary) in the body of the loop has a value within this range.
- **I** is local to the loop, so you cannot refer to **I** outside the loop.
- Although the value of **I** is incremented at each iteration, from the program's perspective it is constant. An attempt to modify its value is illegal; the compiler would reject the program.
- **Integer'Image** is a function that takes an Integer and converts it to a **String**. It is an example of a language construct known as an *attribute*, indicated by the '**'** syntax, which will be covered in more detail later.
- The **&** symbol is the concatenation operator for String values
- The **end loop** marks the end of the loop

The "step" of the loop is limited to 1 (forward direction) and -1 (backward). To iterate backwards over a range, use the **reverse** keyword:

Listing 7: greet_5a_reverse.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet_5a_Reverse is
4    begin
5      for I in reverse 1 .. 5 loop
6          Put_Line ("Hello, World!"
7                      & Integer'Image (I));
8      end loop;
9  end Greet_5a_Reverse;
```

Runtime output

```
Hello, World! 5  
Hello, World! 4  
Hello, World! 3  
Hello, World! 2  
Hello, World! 1
```

The bounds of a **for** loop may be computed at run-time; they are evaluated once, before the loop body is executed. If the value of the upper bound is less than the value of the lower bound, then the loop is not executed at all. This is the case also for **reverse** loops. Thus no output is produced in the following example:

Listing 8: greet_no_op.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;  
2  
3  procedure Greet_No_Op is  
4  begin  
5      for I in reverse 5 .. 1 loop  
6          Put_Line ("Hello, World!"  
7                      & Integer'Image (I));  
8      end loop;  
9  end Greet_No_Op;
```

Build output

```
greet_no_op.adb:5:23: warning: loop range is null, loop will not execute [enabled  
↪ by default]
```

The **for** loop is more general than what we illustrated here; more on that later.

2.3.2 Bare loops

The simplest loop in Ada is the bare loop, which forms the foundation of the other kinds of Ada loops.

Listing 9: greet_5b.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;  
2  
3  procedure Greet_5b is  
4      -- Variable declaration:  
5      I : Integer := 1;  
6      -- ^ Type  
7      --           ^ Initial value  
8  begin  
9      loop  
10         Put_Line ("Hello, World!"  
11                         & Integer'Image (I));  
12  
13         -- Exit statement:  
14         exit when I = 5;  
15         --           ^ Boolean condition  
16  
17         -- Assignment:  
18         I := I + 1;  
19         -- There is no I++ short form to  
20         -- increment a variable  
21     end loop;  
22  end Greet_5b;
```

Runtime output

```
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

This example has the same effect as Greet_5a shown earlier.

It illustrates several concepts:

- We have declared a variable named I between the **is** and the **begin**. This constitutes a *declarative region*. Ada clearly separates the declarative region from the statement part of a subprogram. A declaration can appear in a declarative region but is not allowed as a statement.
- The bare loop statement is introduced by the keyword **loop** on its own and, like every kind of loop statement, is terminated by the combination of keywords **end loop**. On its own, it is an infinite loop. You can break out of it with an **exit** statement.
- The syntax for assignment is **:=**, and the one for equality is **=**. There is no way to confuse them, because as previously noted, in Ada, statements and expressions are distinct, and expressions are not valid statements.

2.3.3 While loops

The last kind of loop in Ada is the **while** loop.

Listing 10: greet_5c.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet_5c is
4      I : Integer := 1;
5  begin
6      -- Condition must be a Boolean value
7      -- (no Integers).
8      -- Operator "<=" returns a Boolean
9      while I <= 5 loop
10         Put_Line ("Hello, World!" & Integer'Image (I));
11
12         I := I + 1;
13     end loop;
14  end Greet_5c;
```

Runtime output

```
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
```

The condition is evaluated before each iteration. If the result is false, then the loop is terminated.

This program has the same effect as the previous examples.

In other languages

Note that Ada has different semantics than C-based languages with respect to the condition in a while loop. In Ada the condition has to be a Boolean value or the compiler will reject the program; the condition is not an integer that is treated as either **True** or **False** depending on whether it is non-zero or zero.

2.4 Imperative language - Case statement

Ada's **case** statement is similar to the C and C++ **switch** statement, but with some important differences.

Here's an example, a variation of a program that was shown earlier with an **if** statement:

Listing 11: check_direction.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Check_Direction is
5      N : Integer;
6  begin
7      loop
8          Put ("Enter an integer value: ");
9          Get (N);
10         Put (N);
11
12         case N is
13             when 0 .. 360 =>
14                 Put_Line (" is due north");
15             when 1 .. 89 =>
16                 Put_Line (" is in the northeast quadrant");
17             when 90 =>
18                 Put_Line (" is due east");
19             when 91 .. 179 =>
20                 Put_Line (" is in the southeast quadrant");
21             when 180 =>
22                 Put_Line (" is due south");
23             when 181 .. 269 =>
24                 Put_Line (" is in the southwest quadrant");
25             when 270 =>
26                 Put_Line (" is due west");
27             when 271 .. 359 =>
28                 Put_Line (" is in the northwest quadrant");
29             when others =>
30                 Put_Line (" Au revoir");
31                 exit;
32         end case;
33     end loop;
34 end Check_Direction;
```

This program repeatedly prompts for an integer value and then, if the value is in the range **0 .. 360**, displays the associated quadrant or axis. If the value is an Integer outside this range, the loop (and the program) terminate after outputting a farewell message.

The effect of the case statement is similar to the if statement in an earlier example, but the case statement can be more efficient because it does not involve multiple range tests.

Notable points about Ada's case statement:

- The case expression (here the variable **N**) must be of a discrete type, i.e. either an integer type or an enumeration type. Discrete types will be covered in more detail

later *discrete types* (page 43).

- Every possible value for the case expression needs to be covered by a unique branch of the case statement. This will be checked at compile time.
- A branch can specify a single value, such as `0`; a range of values, such as `1 .. 89`; or any combination of the two (separated by a `|`).
- As a special case, an optional final branch can specify `others`, which covers all values not included in the earlier branches.
- Execution consists of the evaluation of the case expression and then a transfer of control to the statement sequence in the unique branch that covers that value.
- When execution of the statements in the selected branch has completed, control resumes after the `end case`. Unlike C, execution does not fall through to the next branch. So Ada doesn't need (and doesn't have) a `break` statement.

2.5 Imperative language - Declarative regions

As mentioned earlier, Ada draws a clear syntactic separation between declarations, which introduce names for entities that will be used in the program, and statements, which perform the processing. The areas in the program where declarations may appear are known as declarative regions.

In any subprogram, the section between the `is` and the `begin` is a declarative region. You can have variables, constants, types, inner subprograms, and other entities there.

We've briefly mentioned variable declarations in previous subsection. Let's look at a simple example, where we declare an integer variable `X` in the declarative region and perform an initialization and an addition on it:

Listing 12: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      X : Integer;
5  begin
6      X := 0;
7      Put_Line ("The initial value of X is "
8                 & Integer'Image (X));
9
10     Put_Line ("Performing operation on X... ");
11     X := X + 1;
12
13     Put_Line ("The value of X now is "
14                 & Integer'Image (X));
15 end Main;

```

Runtime output

```

The initial value of X is 0
Performing operation on X...
The value of X now is 1

```

Let's look at an example of a nested procedure:

Listing 13: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     procedure Nested is
5     begin
6         Put_Line ("Hello World");
7     end Nested;
8 begin
9     Nested;
10    -- Call to Nested
11 end Main;
```

Runtime output

```
Hello World
```

A declaration cannot appear as a statement. If you need to declare a local variable amidst the statements, you can introduce a new declarative region with a block statement:

Listing 14: greet.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4 begin
5     loop
6         Put_Line ("Please enter your name: ");
7
8         declare
9             Name : String := Get_Line;
10            -- ^ Call to the
11            --      Get_Line function
12         begin
13             exit when Name = "";
14             Put_Line ("Hi " & Name & "!");
15         end;
16
17         -- Name is undefined here
18     end loop;
19
20     Put_Line ("Bye!");
21 end Greet;
```

Build output

```
greet.adb:9:10: warning: "Name" is not modified, could be declared constant [-gnatwk]
```

Attention: The `Get_Line` function allows you to receive input from the user, and get the result as a string. It is more or less equivalent to the `scanf` C function.

It returns a **String**, which, as we will see later, is an *Unconstrained array type* (page 73). For now we simply note that, if you wish to declare a **String** variable and do not know its size in advance, then you need to initialize the variable during its declaration.

2.6 Imperative language - conditional expressions

Ada 2012 introduced an expression analog for conditional statements (**if** and **case**).

2.6.1 If expressions

Here's an alternative version of an example we saw earlier; the **if** statement has been replaced by an **if** expression:

Listing 15: check_positive.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4  procedure Check_Positive is
5      N : Integer;
6  begin
7      Put ("Enter an integer value: ");
8      Get (N);
9      Put (N);
10
11     declare
12         S : constant String :=
13             (if N > 0 then " is a positive number"
14              else " is not a positive number");
15     begin
16         Put_Line (S);
17     end;
18 end Check_Positive;

```

The **if** expression evaluates to one of the two Strings depending on N, and assigns that value to the local variable S.

Ada's **if** expressions are similar to **if** statements. However, there are a few differences that stem from the fact that it is an expression:

- All branches' expressions must be of the same type
- It *must* be surrounded by parentheses if the surrounding expression does not already contain them
- An **else** branch is mandatory unless the expression following **then** has a Boolean value. In that case an **else** branch is optional and, if not present, defaults to **else True**.

Here's another example:

Listing 16: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4  begin
5      for I in 1 .. 10 loop
6          Put_Line (if I mod 2 = 0 then "Even" else "Odd");
7      end loop;
8  end Main;

```

Runtime output

```
Odd  
Even  
Odd  
Even  
Odd  
Even  
Odd  
Even  
Odd  
Even
```

This program produces 10 lines of output, alternating between "Odd" and "Even".

2.6.2 Case expressions

Analogous to **if** expressions, Ada also has **case** expressions. They work just as you would expect.

Listing 17: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;  
2  
3  procedure Main is  
4  begin  
5      for I in 1 .. 10 loop  
6          Put_Line (case I is  
7              when 1 | 3 | 5 | 7 | 9 => "Odd",  
8              when 2 | 4 | 6 | 8 | 10 => "Even");  
9      end loop;  
10 end Main;
```

Runtime output

```
Odd  
Even  
Odd  
Even  
Odd  
Even  
Odd  
Even  
Odd  
Even
```

This program has the same effect as the preceding example.

The syntax differs from **case** statements, with branches separated by commas.

SUBPROGRAMS

3.1 Subprograms

So far, we have used procedures, mostly to have a main body of code to execute. Procedures are one kind of *subprogram*.

There are two kinds of subprograms in Ada, *functions* and *procedures*. The distinction between the two is that a function returns a value, and a procedure does not.

This example shows the declaration and definition of a function:

Listing 1: increment.ads

```
1 function Increment (I : Integer) return Integer;
```

Listing 2: increment.adb

```
1 -- We declare (but don't define) a function with
2 -- one parameter, returning an integer value
3
4 function Increment (I : Integer) return Integer is
5     -- We define the Increment function
6 begin
7     return I + 1;
8 end Increment;
```

Subprograms in Ada can, of course, have parameters. One syntactically important note is that a subprogram which has no parameters does not have a parameter section at all, for example:

```
procedure Proc;

function Func return Integer;
```

Here's another variation on the previous example:

Listing 3: increment_by.ads

```
1 function Increment_By
2     (I      : Integer := 0;
3      Incr   : Integer := 1) return Integer;
4      -- ^ Default value for parameters
```

In this example, we see that parameters can have default values. When calling the subprogram, you can then omit parameters if they have a default value. Unlike C/C++, a call to a subprogram without parameters does not include parentheses.

This is the implementation of the function above:

Listing 4: increment_by.adb

```

1  function Increment_By
2    (I      : Integer := 0;
3     Incr   : Integer := 1) return Integer is
4 begin
5   return I + Incr;
6 end Increment_By;

```

In the GNAT toolchain

The Ada standard doesn't mandate in which file the specification or the implementation of a subprogram must be stored. In other words, the standard doesn't require a specific file structure or specific file name extensions. For example, we could save both the specification and the implementation of the Increment function above in a file called `increment.txt`. (We could even store the entire source-code of a system in a single file.) From the standard's perspective, this would be completely acceptable.

The GNAT toolchain, however, requires the following file naming scheme:

- files with the `.ads` extension contain the specification, while
- files with the `.adb` extension contain the implementation.

Therefore, in the GNAT toolchain, the specification of the Increment function must be stored in the `increment.ads` file, while its implementation must be stored in the `increment.adb` file. This rule always applies to packages, which we discuss *later* (page 31). (Note, however, that it's possible to circumvent this rule.) For more details, you may refer to the *Introduction to GNAT Toolchain* (page 935) course or the *GPRbuild User's Guide*⁸.

3.1.1 Subprogram calls

We can then call our subprogram this way:

Listing 5: show_increment.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Increment_By;
3
4  procedure Show_Increment is
5    A, B, C : Integer;
6  begin
7    C := Increment_By;
8    --          ^ Parameterless call,
9    --          value of I is 0
10   --          and Incr is 1
11
12   Put_Line ("Using defaults for Increment_By is "
13             & Integer'Image (C));
14
15   A := 10;
16   B := 3;
17   C := Increment_By (A, B);
18   --          ^ Regular parameter passing
19
20   Put_Line ("Increment of "
21             & Integer'Image (A)

```

(continues on next page)

⁸ https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html

(continued from previous page)

```

22      & " with "
23      & Integer'Image (B)
24      & " is "
25      & Integer'Image (C));
26
27  A := 20;
28  B := 5;
29  C := Increment_By (I    => A,
30                      Incr => B);
31          --           ^ Named parameter passing
32
33  Put_Line ("Increment of "
34            & Integer'Image (A)
35            & " with "
36            & Integer'Image (B)
37            & " is "
38            & Integer'Image (C));
39 end Show_Increment;

```

Runtime output

```

Using defaults for Increment_By is 1
Increment of 10 with 3 is 13
Increment of 20 with 5 is 25

```

Ada allows you to name the parameters when you pass them, whether they have a default or not. There are some rules:

- Positional parameters come first.
- A positional parameter cannot follow a named parameter.

As a convention, people usually name parameters at the call site if the function's corresponding parameters has a default value. However, it is also perfectly acceptable to name every parameter if it makes the code clearer.

3.1.2 Nested subprograms

As briefly mentioned earlier, Ada allows you to declare one subprogram inside of another.

This is useful for two reasons:

- It lets you organize your programs in a cleaner fashion. If you need a subprogram only as a "helper" for another subprogram, then the principle of localization indicates that the helper subprogram should be declared nested.
- It allows you to share state easily in a controlled fashion, because the nested subprograms have access to the parameters, as well as any local variables, declared in the outer scope.

For the previous example, we can move the duplicated code (call to Put_Line) to a separate procedure. This is a shortened version with the nested Display_Result procedure.

Listing 6: show_increment.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Increment_By;
3
4  procedure Show_Increment is
5    A, B, C : Integer;
6

```

(continues on next page)

(continued from previous page)

```

7  procedure Display_Result is
8  begin
9      Put_Line ("Increment of "
10         & Integer'Image (A)
11         & " with "
12         & Integer'Image (B)
13         & " is "
14         & Integer'Image (C));
15  end Display_Result;
16
17 begin
18     A := 10;
19     B := 3;
20     C := Increment_By (A, B);
21     Display_Result;
22     A := 20;
23     B := 5;
24     C := Increment_By (A, B);
25     Display_Result;
26 end Show_Increment;

```

Runtime output

```

Increment of 10 with 3 is 13
Increment of 20 with 5 is 25

```

3.1.3 Function calls

An important feature of function calls in Ada is that the return value at a call cannot be ignored; that is, a function call cannot be used as a statement.

If you want to call a function and do not need its result, you will still need to explicitly store it in a local variable.

Listing 7: quadruple.adb

```

1  function Quadruple (I : Integer) return Integer is
2      function Double (I : Integer) return Integer is
3          begin
4              return I * 2;
5          end Double;
6
7      Res : Integer := Double (Double (I));
8          -- ^ Calling the Double
9          -- function
10
11 begin
12     Double (I);
13     -- ERROR: cannot use call to function
14     -- "Double" as a statement
15
16     return Res;
end Quadruple;

```

Build output

```

quadruple.adb:11:04: error: cannot use call to function "Double" as a statement
quadruple.adb:11:04: error: return value of a function call cannot be ignored
gprbuild: *** compilation phase failed

```

In the GNAT toolchain

In GNAT, with all warnings activated, it becomes even harder to ignore the result of a function, because unused variables will be flagged. For example, this code would not be valid:

```
function Read_Int
  (Stream : Network_Stream;
   Result : out Integer) return Boolean;

procedure Main is
  Stream : Network_Stream := Get_Stream;
  My_Int : Integer;

  -- Warning: in the line below, B is
  --           never read.
  B : Boolean := Read_Int (Stream, My_Int);
begin
  null;
end Main;
```

You then have two solutions to silence this warning:

- Either annotate the variable with pragma Unreferenced, thus:

```
B : Boolean := Read_Int (Stream, My_Int);
pragma Unreferenced (B);
```

- Or give the variable a name that contains any of the strings *discard* *dummy* *ignore* *junk* *unused* (case insensitive)
-

3.2 Parameter modes

So far we have seen that Ada is a safety-focused language. There are many ways this is realized, but two important points are:

- Ada makes the user specify as much as possible about the behavior expected for the program, so that the compiler can warn or reject if there is an inconsistency.
- Ada provides a variety of techniques for achieving the generality and flexibility of pointers and dynamic memory management, but without the latter's drawbacks (such as memory leakage and dangling references).

Parameter modes are a feature that helps achieve the two design goals above. A subprogram parameter can be specified with a mode, which is one of the following:

in	Parameter can only be read, not written
out	Parameter can be written to, then read
in out	Parameter can be both read and written

The default mode for parameters is **in**; so far, most of the examples have been using **in** parameters.

Historically

Functions and procedures were originally more different in philosophy. Before Ada 2012, functions could only take **in** parameters.

3.3 Subprogram calls

3.3.1 In parameters

The first mode for parameters is the one we have been implicitly using so far. Parameters passed using this mode cannot be modified, so that the following program will cause an error:

Listing 8: swap.adb

```
1 procedure Swap (A, B : Integer) is
2     Tmp : Integer;
3 begin
4     Tmp := A;
5
6     -- Error: assignment to "in" mode
7     -- parameter not allowed
8     A := B;
9
10    -- Error: assignment to "in" mode
11    -- parameter not allowed
12    B := Tmp;
13 end Swap;
```

Build output

```
swap.adb:8:04: error: assignment to "in" mode parameter not allowed
swap.adb:12:04: error: assignment to "in" mode parameter not allowed
gprbuild: *** compilation phase failed
```

The fact that this is the default mode is in itself very important. It means that a parameter will not be modified unless you explicitly specify a mode in which modification is allowed.

3.3.2 In out parameters

To correct our code above, we can use an **in out** parameter.

Listing 9: in_out_params.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure In_Out_Params is
4     procedure Swap (A, B : in out Integer) is
5         Tmp : Integer;
6     begin
7         Tmp := A;
8         A := B;
9         B := Tmp;
10    end Swap;
11
12    A : Integer := 12;
13    B : Integer := 44;
14 begin
15     Swap (A, B);
16
17     -- Prints 44
18     Put_Line (Integer'Image (A));
19 end In_Out_Params;
```

Runtime output

44

An `in out` parameter will allow read and write access to the object passed as parameter, so in the example above, we can see that A is modified after the call to Swap.

Attention: While `in out` parameters look a bit like references in C++, or regular parameters in Java that are passed by-reference, the Ada language standard does not mandate "by reference" passing for `in out` parameters except for certain categories of types as will be explained later.

In general, it is better to think of modes as higher level than by-value versus by-reference semantics. For the compiler, it means that an array passed as an `in` parameter might be passed by reference, because it is more efficient (which does not change anything for the user since the parameter is not assignable). However, a parameter of a discrete type will always be passed by copy, regardless of its mode (which is more efficient on most architectures).

3.3.3 Out parameters

The `out` mode applies when the subprogram needs to write to a parameter that might be uninitialized at the point of call. Reading the value of an `out` parameter is permitted, but it should only be done after the subprogram has assigned a value to the parameter. Out parameters behave a bit like return values for functions. When the subprogram returns, the actual parameter (a variable) will have the value of the out parameter at the point of return.

In other languages

Ada doesn't have a tuple construct and does not allow returning multiple values from a subprogram (except by declaring a full-fledged record type). Hence, a way to return multiple values from a subprogram is to use out parameters.

For example, a procedure reading integers from the network could have one of the following specifications:

```
procedure Read_Int
  (Stream : Network_Stream;
   Success : out Boolean;
   Result : out Integer);

function Read_Int
  (Stream : Network_Stream;
   Result : out Integer) return Boolean;
```

While reading an out variable before writing to it should, ideally, trigger an error, imposing that as a rule would cause either inefficient run-time checks or complex compile-time rules. So from the user's perspective an out parameter acts like an uninitialized variable when the subprogram is invoked.

In the GNAT toolchain

GNAT will detect simple cases of incorrect use of out parameters. For example, the compiler will emit a warning for the following program:

Listing 10: outp.adb

```

1  procedure Outp is
2      procedure Foo (A : out Integer) is
3          B : Integer := A;
4          --           ^ Warning on reference
5          --           to uninitialized A
6      begin
7          A := B;
8      end Foo;
9      begin
10         null;
11     end Outp;

```

Build output

```

outp.adb:2:14: warning: procedure "Foo" is not referenced [-gnatwu]
outp.adb:3:07: warning: "B" is not modified, could be declared constant [-gnatwk]
outp.adb:3:22: warning: "A" may be referenced before it has a value [enabled by
  default]

```

3.3.4 Forward declaration of subprograms

As we saw earlier, a subprogram can be declared without being fully defined. This is possible in general, and can be useful if you need subprograms to be mutually recursive, as in the example below:

Listing 11: mutually_recursive_subprograms.adb

```

1  procedure Mutually_Recursive_Subprograms is
2      procedure Compute_A (V : Natural);
3      -- Forward declaration of Compute_A
4
5      procedure Compute_B (V : Natural) is
6      begin
7          if V > 5 then
8              Compute_A (V - 1);
9              -- Call to Compute_A
10         end if;
11     end Compute_B;
12
13     procedure Compute_A (V : Natural) is
14     begin
15         if V > 2 then
16             Compute_B (V - 1);
17             -- Call to Compute_B
18         end if;
19     end Compute_A;
20
21     begin
22         Compute_A (15);
23     end Mutually_Recursive_Subprograms;

```

3.4 Renaming

Subprograms can be renamed by using the `renames` keyword and declaring a new name for a subprogram:

```
procedure New_Proc renames Original_Proc;
```

This can be useful, for example, to improve the readability of your application when you're using code from external sources that cannot be changed in your system. Let's look at an example:

Listing 12: a_procedure_with_very_long_name_that_cannot_be_changed.ads

```
1 procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
2   (A_Message : String);
```

Listing 13: a_procedure_with_very_long_name_that_cannot_be_changed.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed
4   (A_Message : String) is
5 begin
6   Put_Line (A_Message);
7 end A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
```

As the wording in the name of procedure above implies, we cannot change its name. We can, however, rename it to something like `Show` in our test application and use this shorter name. Note that we also have to declare all parameters of the original subprogram — we may rename them, too, in the declaration. For example:

Listing 14: show_renaming.adb

```
1 with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
2
3 procedure Show_Renaming is
4
5   procedure Show (S : String) renames
6     A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;
7
8 begin
9   Show ("Hello World!");
10 end Show_Renaming;
```

Runtime output

```
Hello World!
```

Note that the original name (`A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed`) is still visible after the declaration of the `Show` procedure.

We may also rename subprograms from the standard library. For example, we may rename `Integer'Image` to `Img`:

Listing 15: show_image_renaming.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Image_Renaming is
```

(continues on next page)

(continued from previous page)

```
5  function Img (I : Integer) return String
6    renames Integer'Image;
7
8 begin
9   Put_Line (Img (2));
10  Put_Line (Img (3));
11 end Show_Image_Renaming;
```

Runtime output

```
2
3
```

Renaming also allows us to introduce default expressions that were not available in the original declaration. For example, we may specify "Hello World!" as the default for the **String** parameter of the Show procedure:

```
with A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

procedure Show_Renaming_Defaults is

  procedure Show (S : String := "Hello World!")
    renames
      A_Procedure_With_Very_Long_Name_That_Cannot_Be_Changed;

begin
  Show;
end Show_Renaming_Defaults;
```

MODULAR PROGRAMMING

So far, our examples have been simple standalone subprograms. Ada is helpful in that regard, since it allows arbitrary declarations in a declarative part. We were thus able to declare our types and variables in the bodies of main procedures.

However, it is easy to see that this is not going to scale up for real-world applications. We need a better way to structure our programs into modular and distinct units.

Ada encourages the separation of programs into multiple packages and sub-packages, providing many tools to a programmer on a quest for a perfectly organized code-base.

4.1 Packages

Here is an example of a package declaration in Ada:

Listing 1: week.ads

```
1 package Week is
2
3     Mon : constant String := "Monday";
4     Tue : constant String := "Tuesday";
5     Wed : constant String := "Wednesday";
6     Thu : constant String := "Thursday";
7     Fri : constant String := "Friday";
8     Sat : constant String := "Saturday";
9     Sun : constant String := "Sunday";
10
11 end Week;
```

And here is how you use it:

Listing 2: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Week;
3 -- References the Week package, and
4 -- adds a dependency from Main to Week
5
6 procedure Main is
7 begin
8     Put_Line ("First day of the week is "
9               & Week.Mon);
10 end Main;
```

Runtime output

```
First day of the week is Monday
```

Packages let you make your code modular, separating your programs into semantically significant units. Additionally the separation of a package's specification from its body (which we will see below) can reduce compilation time.

While the `with` clause indicates a dependency, you can see in the example above that you still need to prefix the referencing of entities from the `Week` package by the name of the package. (If we had included a `use Week` clause, then such a prefix would not have been necessary.)

Accessing entities from a package uses the dot notation, `A.B`, which is the same notation as the one used to access record fields.

A `with` clause can *only* appear in the prelude of a compilation unit (i.e., before the reserved word, such as `procedure`, that marks the beginning of the unit). It is not allowed anywhere else. This rule is only needed for methodological reasons: the person reading your code should be able to see immediately which units the code depends on.

In other languages

Packages look similar to, but are semantically very different from, header files in C/C++.

- The first and most important distinction is that packages are a language-level mechanism. This is in contrast to a `#include`'d header file, which is a functionality of the C preprocessor.
- An immediate consequence is that the `with` construct is a semantic inclusion mechanism, not a text inclusion mechanism. Hence, when you `with` a package, you are saying to the compiler "I'm depending on this semantic unit", and not "include this bunch of text in place here".
- The effect of a package thus does not vary depending on where it has been `with`ed from. Contrast this with C/C++, where the meaning of the included text depends on the context in which the `#include` appears.

This allows compilation/recompilation to be more efficient. It also allows tools like IDEs to have correct information about the semantics of a program. In turn, this allows better tooling in general, and code that is more analyzable, even by humans.

An important benefit of Ada `with` clauses when compared to `#include` is that it is stateless. The order of `with` and `use` clauses does not matter, and can be changed without side effects.

In the GNAT toolchain

The Ada language standard does not mandate any particular relationship between source files and packages; for example, in theory you can put all your code in one file, or use your own file naming conventions. In practice, however, an implementation will have specific rules. With GNAT, each top-level compilation unit needs to go into a separate file. In the example above, the `Week` package will be in an `.ads` file (for Ada specification), and the `Main` procedure will be in an `.adb` file (for Ada body).

4.2 Using a package

As we have seen above, the `with` clause indicates a dependency on another package. However, every reference to an entity coming from the `Week` package had to be prefixed by the full name of the package. It is possible to make every entity of a package visible directly in the current scope, using the `use` clause.

In fact, we have been using the `use` clause since almost the beginning of this tutorial.

Listing 3: main.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2   -- ^ Make every entity of the
3   -- Ada.Text_Io package
4   -- directly visible.
5 with Week;
6
7 procedure Main is
8   use Week;
9   -- Make every entity of the Week
10  -- package directly visible.
11 begin
12   Put_Line ("First day of the week is " & Mon);
13 end Main;
```

Runtime output

```
First day of the week is Monday
```

As you can see in the example above:

- `Put_Line` is a subprogram that comes from the `Ada.Text_Io` package. We can reference it directly because we have `used` the package at the top of the `Main` unit.
- Unlike `with` clauses, a `use` clause can be placed either in the prelude, or in any declarative region. In the latter case the `use` clause will have an effect in its containing lexical scope.

4.3 Package body

In the simple example above, the `Week` package only has declarations and no body. That's not a mistake: in a package specification, which is what is illustrated above, you cannot declare bodies. Those have to be in the package body.

Listing 4: operations.ads

```

1 package Operations is
2
3   -- Declaration
4   function Increment_By
5     (I      : Integer;
6      Incr : Integer := 0) return Integer;
7
8   function Get_Increment_Value return Integer;
9
10 end Operations;
```

Listing 5: operations.adb

```

1 package body Operations is
2
3     Last_Increment : Integer := 1;
4
5     function Increment_By
6         (I      : Integer;
7          Incr : Integer := 0) return Integer is
8 begin
9     if Incr /= 0 then
10        Last_Increment := Incr;
11    end if;
12
13    return I + Last_Increment;
14 end Increment_By;
15
16     function Get_Increment_Value return Integer is
17 begin
18     return Last_Increment;
19 end Get_Increment_Value;
20
21 end Operations;

```

Here we can see that the body of the Increment_By function has to be declared in the body. Coincidentally, introducing a body allows us to put the Last_Increment variable in the body, and make them inaccessible to the user of the Operations package, providing a first form of encapsulation.

This works because entities declared in the body are *only* visible in the body.

This example shows how Last_Increment is used indirectly:

Listing 6: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Operations;
3
4 procedure Main is
5     use Operations;
6
7     I : Integer := 0;
8     R : Integer;
9
10    procedure Display_Update_Values is
11        Incr : constant Integer := Get_Increment_Value;
12    begin
13        Put_Line (Integer'Image (I)
14                  & " incremented by "
15                  & Integer'Image (Incr)
16                  & " is "
17                  & Integer'Image (R));
18        I := R;
19    end Display_Update_Values;
20 begin
21    R := Increment_By (I);
22    Display_Update_Values;
23    R := Increment_By (I);
24    Display_Update_Values;
25
26    R := Increment_By (I, 5);
27    Display_Update_Values;

```

(continues on next page)

(continued from previous page)

```

28   R := Increment_By (I);
29   Display_Update_Values;
30
31   R := Increment_By (I, 10);
32   Display_Update_Values;
33   R := Increment_By (I);
34   Display_Update_Values;
35 end Main;
```

Runtime output

```

0 incremented by 1 is 1
1 incremented by 1 is 2
2 incremented by 5 is 7
7 incremented by 5 is 12
12 incremented by 10 is 22
22 incremented by 10 is 32
```

4.4 Child packages

Packages can be used to create hierarchies. We achieve this by using child packages, which extend the functionality of their parent package. One example of a child package that we've been using so far is the Ada.Text_Io package. Here, the parent package is called Ada, while the child package is called Text_Io. In the previous examples, we've been using the Put_Line procedure from the Text_Io child package.

Important

Ada also supports nested packages. However, since they can be more complicated to use, the recommendation is to use child packages instead. Nested packages will be covered in the advanced course.

Let's begin our discussion on child packages by taking our previous Week package:

Listing 7: week.ads

```

1 package Week is
2
3   Mon : constant String := "Monday";
4   Tue : constant String := "Tuesday";
5   Wed : constant String := "Wednesday";
6   Thu : constant String := "Thursday";
7   Fri : constant String := "Friday";
8   Sat : constant String := "Saturday";
9   Sun : constant String := "Sunday";
10
11 end Week;
```

If we want to create a child package for Week, we may write:

Listing 8: week-child.ads

```

1 package Week.Child is
2
3   function Get_First_Of_Week return String;
4
5 end Week.Child;
```

Here, Week is the parent package and Child is the child package. This is the corresponding package body of Week.Child:

Listing 9: week-child.adb

```
1 package body Week.Child is
2
3     function Get_First_Of_Week return String is
4     begin
5         return Mon;
6     end Get_First_Of_Week;
7
8 end Week.Child;
```

In the implementation of the Get_First_of_Week function, we can use the Mon string directly, even though it was declared in the parent package Week. We don't write `with` Week here because all elements from the specification of the Week package — such as Mon, Tue and so on — are visible in the child package Week.Child.

Now that we've completed the implementation of the Week.Child package, we can use elements from this child package in a subprogram by simply writing `with` Week.Child. Similarly, if we want to use these elements directly, we write `use` Week.Child in addition. For example:

Listing 10: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Week.Child;  use Week.Child;
3
4 procedure Main is
5 begin
6     Put_Line ("First day of the week is "
7               & Get_First_of_Week);
8 end Main;
```

Runtime output

```
First day of the week is Monday
```

4.4.1 Child of a child package

So far, we've seen a two-level package hierarchy. But the hierarchy that we can potentially create isn't limited to that. For instance, we could extend the hierarchy of the previous source-code example by declaring a Week.Child.Grandchild package. In this case, Week.Child would be the parent of the Grandchild package. Let's consider this implementation:

Listing 11: week-child-grandchild.ads

```
1 package Week.Child.Grandchild is
2
3     function Get_Second_of_Week return String;
4
5 end Week.Child.Grandchild;
```

Listing 12: week-child-grandchild.adb

```
1 package body Week.Child.Grandchild is
2
3     function Get_Second_of_Week return String is
```

(continues on next page)

(continued from previous page)

```

4 begin
5   return Tue;
6 end Get_Second_Of_Week;
7
8 end Week.Child.Grandchild;

```

We can use this new Grandchild package in our test application in the same way as before: we can reuse the previous test application and adapt the `with` and `use`, and the function call. This is the updated code:

Listing 13: main.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Week.Child.Grandchild; use Week.Child.Grandchild;
3
4 procedure Main is
5 begin
6   Put_Line ("Second day of the week is "
7             & Get_Second_Of_Week);
8 end Main;

```

Runtime output

```
Second day of the week is Tuesday
```

Again, this isn't the limit for the package hierarchy. We could continue to extend the hierarchy of the previous example by implementing a `Week.Child.Grandchild.Grand_child` package.

4.4.2 Multiple children

So far, we've seen a single child package of a parent package. However, a parent package can also have multiple children. We could extend the example above and implement a `Week.Child_2` package. For example:

Listing 14: week-child_2.ads

```

1 package Week.Child_2 is
2
3   function Get_Last_Of_Week return String;
4
5 end Week.Child_2;

```

Here, `Week` is still the parent package of the `Child` package, but it's also the parent of the `Child_2` package. In the same way, `Child_2` is obviously one of the child packages of `Week`.

This is the corresponding package body of `Week.Child_2`:

Listing 15: week-child_2.adb

```

1 package body Week.Child_2 is
2
3   function Get_Last_Of_Week return String is
4     begin
5       return Sun;
6     end Get_Last_Of_Week;
7
8 end Week.Child_2;

```

We can now reference both children in our test application:

Listing 16: main.adb

```
1 with Ada.Text_IO;  use Ada.Text_IO;
2 with Week.Child;   use Week.Child;
3 with Week.Child_2; use Week.Child_2;
4
5 procedure Main is
6 begin
7   Put_Line ("First day of the week is "
8             & Get_First_Of_Week);
9   Put_Line ("Last day of the week is "
10            & Get_Last_Of_Week);
11 end Main;
```

Runtime output

```
First day of the week is Monday
Last day of the week is Sunday
```

4.4.3 Visibility

In the previous section, we've seen that elements declared in a parent package specification are visible in the child package. This is, however, not the case for elements declared in the package body of a parent package.

Let's consider the package Book and its child Additional_Operations:

Listing 17: book.ads

```
1 package Book is
2
3   Title : constant String := "Visible for my children";
4
5   function Get_Title return String;
6
7   function Get_Author return String;
8
9
10 end Book;
```

Listing 18: book-additional_operations.ads

```
1 package Book.Additional_Operations is
2
3   function Get_Extended_Title return String;
4
5   function Get_Extended_Author return String;
6
7 end Book.Additional_Operations;
```

This is the body of both packages:

Listing 19: book.adb

```
1 package body Book is
2
3   Author : constant String := "Author not visible for my children";
4
5
```

(continues on next page)

(continued from previous page)

```

6   function Get_Title return String is
7   begin
8     return Title;
9   end Get_Title;
10
11  function Get_Author return String is
12  begin
13    return Author;
14  end Get_Author;
15
16 end Book;

```

Listing 20: book-additional_operations.adb

```

1 package body Book.Additional_Operations is
2
3   function Get_Extended_Title return String is
4   begin
5     return "Book Title: " & Title;
6   end Get_Extended_Title;
7
8   function Get_Extended_Author return String is
9   begin
10    -- "Author" string declared in the body
11    -- of the Book package is not visible
12    -- here. Therefore, we cannot write:
13    --
14    -- return "Book Author: " & Author;
15
16    return "Book Author: Unknown";
17  end Get_Extended_Author;
18
19 end Book.Additional_Operations;

```

In the implementation of the Get_Extended_Title, we're using the Title constant from the parent package Book. However, as indicated in the comments of the Get_Extended_Author function, the Author string — which we declared in the body of the Book package — isn't visible in the Book.Additional_Operations package. Therefore, we cannot use it to implement the Get_Extended_Author function.

We can, however, use the Get_Author function from Book in the implementation of the Get_Extended_Author function to retrieve this string. Likewise, we can use this strategy to implement the Get_Extended_Title function. This is the adapted code:

Listing 21: book-additional_operations.adb

```

1 package body Book.Additional_Operations is
2
3   function Get_Extended_Title return String is
4   begin
5     return "Book Title: " & Get_Title;
6   end Get_Extended_Title;
7
8   function Get_Extended_Author return String is
9   begin
10    return "Book Author: " & Get_Author;
11  end Get_Extended_Author;
12
13 end Book.Additional_Operations;

```

This is a simple test application for the packages above:

Listing 22: main.adb

```
1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Book.Additional_Operations; use Book.Additional_Operations;
3
4 procedure Main is
5 begin
6   Put_Line (Get_Extended_Title);
7   Put_Line (Get_Extended_Author);
8 end Main;
```

Runtime output

```
Book Title: Visible for my children
Book Author: Author not visible for my children
```

By declaring elements in the body of a package, we can implement encapsulation in Ada. Those elements will only be visible in the package body, but nowhere else. This isn't, however, the only way to achieve encapsulation in Ada: we'll discuss other approaches in the *Privacy* (page 107) chapter.

4.5 Renaming

Previously, we've mentioned that *subprograms can be renamed* (page 29). We can rename packages, too. Again, we use the `renames` keyword for that. The following example renames the `Ada.Text_IO` package as `TI0`:

Listing 23: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4   package TI0 renames Ada.Text_IO;
5 begin
6   TI0.Put_Line ("Hello");
7 end Main;
```

Runtime output

```
Hello
```

We can use renaming to improve the readability of our code by using shorter package names. In the example above, we write `TI0.Put_Line` instead of the longer version (`Ada.Text_IO.Put_Line`). This approach is especially useful when we don't `use` packages and want to avoid that the code becomes too verbose.

Note we can also rename subprograms and objects inside packages. For instance, we could have just renamed the `Put_Line` procedure in the source-code example above:

Listing 24: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4   procedure Say (Something : String)
5     renames Ada.Text_IO.Put_Line;
6 begin
7   Say ("Hello");
8 end Main;
```

Runtime output

Hello

STRONGLY TYPED LANGUAGE

Ada is a strongly typed language. It is interestingly modern in that respect: strong static typing has been increasing in popularity in programming language design, owing to factors such as the growth of statically typed functional programming, a big push from the research community in the typing domain, and many practical languages with strong type systems.

5.1 What is a type?

In statically typed languages, a type is mainly (but not only) a *compile time* construct. It is a construct to enforce invariants about the behavior of a program. Invariants are unchangeable properties that hold for all variables of a given type. Enforcing them ensures, for example, that variables of a data type never have invalid values.

A type is used to reason about the *objects* a program manipulates (an object is a variable or a constant). The aim is to classify objects by what you can accomplish with them (i.e., the operations that are permitted), and this way you can reason about the correctness of the objects' values.

5.2 Integers

A nice feature of Ada is that you can define your own integer types, based on the requirements of your program (i.e., the range of values that makes sense). In fact, the definitional mechanism that Ada provides forms the semantic basis for the predefined integer types. There is no "magical" built-in type in that regard, which is unlike most languages, and arguably very elegant.

Listing 1: integer_type_example.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Integer_Type_Example is
4      -- Declare a signed integer type,
5      -- and give the bounds
6      type My_Int is range -1 .. 20;
7          --                                ^ High bound
8          --                                ^ Low bound
9
10     -- Like variables, type declarations can
11     -- only appear in declarative regions.
12 begin
13     for I in My_Int loop
14         Put_Line (My_Int'Image (I));
15         --                                ^ 'Image attribute
```

(continues on next page)

(continued from previous page)

```
16      -- converts a value
17      -- to a String.
18  end loop;
19 end Integer_Type_Example;
```

Runtime output

```
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

This example illustrates the declaration of a signed integer type, and several things we can do with them.

Every type declaration in Ada starts with the **type** keyword (except for *task types* (page 152)). After the type, we can see a range that looks a lot like the ranges that we use in for loops, that defines the low and high bound of the type. Every integer in the inclusive range of the bounds is a valid value for the type.

Ada integer types

In Ada, an integer type is not specified in terms of its machine representation, but rather by its range. The compiler will then choose the most appropriate representation.

Another point to note in the above example is the `My_Int'Image (I)` expression. The `Name'Attribute (optional params)` notation is used for what is called an attribute in Ada. An attribute is a built-in operation on a type, a value, or some other program entity. It is accessed by using a '`'` symbol (the ASCII apostrophe).

Ada has several types available as "built-ins"; **Integer** is one of them. Here is how **Integer** might be defined for a typical processor:

```
type Integer is
  range -(2 ** 31) .. +(2 ** 31 - 1);
```

`**` is the exponent operator, which means that the first valid value for **Integer** is -2^{31} , and the last valid value is $2^{31} - 1$.

Ada does not mandate the range of the built-in type **Integer**. An implementation for a 16-bit target would likely choose the range -2^{15} through $2^{15} - 1$.

5.2.1 Operational semantics

Unlike some other languages, Ada requires that operations on integers should be checked for overflow.

Listing 2: main.adb

```

1 procedure Main is
2   A : Integer := Integer'Last;
3   B : Integer;
4 begin
5   B := A + 5;
6   -- This operation will overflow, eg. it
7   -- will raise an exception at run time.
8 end Main;

```

Build output

```

main.adb:2:04: warning: "A" is not modified, could be declared constant [-gnatwk]
main.adb:3:04: warning: variable "B" is assigned but never read [-gnatwm]
main.adb:5:04: warning: possibly useless assignment to "B", value might not be referenced [-gnatwm]
main.adb:5:11: warning: value not in range of type "Standard.Integer" [enabled by default]
main.adb:5:11: warning: Constraint_Error will be raised at run time [enabled by default]

```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:5 overflow check failed
```

There are two types of overflow checks:

- Machine-level overflow, when the result of an operation exceeds the maximum value (or is less than the minimum value) that can be represented in the storage reserved for an object of the type, and
- Type-level overflow, when the result of an operation is outside the range defined for the type.

Mainly for efficiency reasons, while machine level overflow always results in an exception, type level overflows will only be checked at specific boundaries, like assignment:

Listing 3: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type My_Int is range 1 .. 20;
5   A : My_Int := 12;
6   B : My_Int := 15;
7   M : My_Int := (A + B) / 2;
8   -- No overflow here, overflow checks
9   -- are done at specific boundaries.
10 begin
11   for I in 1 .. M loop
12     Put_Line ("Hello, World!");
13   end loop;
14   -- Loop body executed 13 times
15 end Main;

```

Build output

```
main.adb:5:04: warning: "A" is not modified, could be declared constant [-gnatwk]
main.adb:6:04: warning: "B" is not modified, could be declared constant [-gnatwk]
main.adb:7:04: warning: "M" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Hello, World!
```

Type level overflow will only be checked at specific points in the execution. The result, as we see above, is that you might have an operation that overflows in an intermediate computation, but no exception will be raised because the final result does not overflow.

5.3 Unsigned types

Ada also features unsigned Integer types. They're called *modular* types in Ada parlance. The reason for this designation is due to their behavior in case of overflow: They simply "wrap around", as if a modulo operation was applied.

For machine sized modular types, for example a modulus of 2^{32} , this mimics the most common implementation behavior of unsigned types. However, an advantage of Ada is that the modulus is more general:

Listing 4: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Mod_Int is mod 2 ** 5;
5      --          ^ Range is 0 .. 31
6
7      A : constant Mod_Int := 20;
8      B : constant Mod_Int := 15;
9
10     M : constant Mod_Int := A + B;
11     -- No overflow here,
12     -- M = (20 + 15) mod 32 = 3
13 begin
14     for I in 1 .. M loop
15         Put_Line ("Hello, World!");
16     end loop;
17 end Main;
```

Runtime output

```
Hello, World!
Hello, World!
Hello, World!
```

Unlike in C/C++, since this wraparound behavior is guaranteed by the Ada specification, you can rely on it to implement portable code. Also, being able to leverage the wrapping on arbitrary bounds is very useful — the modulus does not need to be a power of 2 — to implement certain algorithms and data structures, such as [ring buffers](#)⁹.

5.4 Enumerations

Enumeration types are another nicety of Ada's type system. Unlike C's enums, they are *not* integers, and each new enumeration type is incompatible with other enumeration types. Enumeration types are part of the bigger family of discrete types, which makes them usable in certain situations that we will describe later but one context that we have already seen is a case statement.

Listing 5: enumeration_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Enumeration_Example is
4      type Days is (Monday, Tuesday, Wednesday,
5                     Thursday, Friday,
6                     Saturday, Sunday);
7      -- An enumeration type
8  begin
9      for I in Days loop
10         case I is
11             when Saturday .. Sunday =>
12                 Put_Line ("Week end!");
13
14             when Monday .. Friday =>
15                 Put_Line ("Hello on "
16                           & Days'Image (I));
17                 -- 'Image attribute, works on
18                 -- enums too
19         end case;
20     end loop;
21  end Enumeration_Example;
```

Runtime output

```
Hello on MONDAY
Hello on TUESDAY
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
Week end!
Week end!
```

Enumeration types are powerful enough that, unlike in most languages, they're used to define the standard Boolean type:

```
type Boolean is (False, True);
```

As mentioned previously, every "built-in" type in Ada is defined with facilities generally available to the user.

⁹ https://en.wikipedia.org/wiki/Circular_buffer

5.5 Floating-point types

5.5.1 Basic properties

Like most languages, Ada supports floating-point types. The most commonly used floating-point type is **Float**:

Listing 6: floating_point_demo.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Floating_Point_Demo is
4     A : constant Float := 2.5;
5 begin
6     Put_Line ("The value of A is "
7                & Float'Image (A));
8 end Floating_Point_Demo;
```

Runtime output

```
The value of A is 2.50000E+00
```

The application will display **2.5** as the value of A.

The Ada language does not specify the precision (number of decimal digits in the mantissa) for **Float**; on a typical 32-bit machine the precision will be 6.

All common operations that could be expected for floating-point types are available, including absolute value and exponentiation. For example:

Listing 7: floating_point_operations.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Floating_Point_Operations is
4     A : Float := 2.5;
5 begin
6     A := abs (A - 4.5);
7     Put_Line ("The value of A is "
8                & Float'Image (A));
9
10    A := A ** 2 + 1.0;
11    Put_Line ("The value of A is "
12                & Float'Image (A));
13 end Floating_Point_Operations;
```

Runtime output

```
The value of A is 2.00000E+00
The value of A is 5.00000E+00
```

The value of A is **2.0** after the first operation and **5.0** after the second operation.

In addition to **Float**, an Ada implementation may offer data types with higher precision such as **Long_Float** and **Long_Long_Float**. Like **Float**, the standard does not indicate the exact precision of these types: it only guarantees that the type **Long_Float**, for example, has at least the precision of **Float**. In order to guarantee that a certain precision requirement is met, we can define custom floating-point types, as we will see in the next section.

5.5.2 Precision of floating-point types

Ada allows the user to specify the precision for a floating-point type, expressed in terms of decimal digits. Operations on these custom types will then have at least the specified precision. The syntax for a simple floating-point type declaration is:

```
type T is digits <number_of_decimal_digits>;
```

The compiler will choose a floating-point representation that supports the required precision. For example:

Listing 8: custom_floating_types.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Custom_Floating_Types is
4     type T3 is digits 3;
5     type T15 is digits 15;
6     type T18 is digits 18;
7 begin
8     Put_Line ("T3 requires "
9                 & Integer'Image (T3'Size)
10                & " bits");
11     Put_Line ("T15 requires "
12                 & Integer'Image (T15'Size)
13                & " bits");
14     Put_Line ("T18 requires "
15                 & Integer'Image (T18'Size)
16                & " bits");
17 end Custom_Floating_Types;
```

Runtime output

```
T3 requires 32 bits
T15 requires 64 bits
T18 requires 128 bits
```

In this example, the attribute '`Size`' is used to retrieve the number of bits used for the specified data type. As we can see by running this example, the compiler allocates 32 bits for T3, 64 bits for T15 and 128 bits for T18. This includes both the mantissa and the exponent.

The number of digits specified in the data type is also used in the format when displaying floating-point variables. For example:

Listing 9: display_custom_floating_types.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Display_Custom_Floating_Types is
4     type T3 is digits 3;
5     type T18 is digits 18;
6
7     C1 : constant := 1.0e-4;
8
9     A : constant T3 := 1.0 + C1;
10    B : constant T18 := 1.0 + C1;
11 begin
12     Put_Line ("The value of A is "
13                 & T3'Image (A));
14     Put_Line ("The value of B is "
```

(continues on next page)

(continued from previous page)

```
15      & T18'Image (B));
16 end Display_Custom_Floating_Types;
```

Runtime output

```
The value of A is 1.00E+00
The value of B is 1.0001000000000000E+00
```

As expected, the application will display the variables according to specified precision (1.00E+00 and 1.0001000000000000E+00).

5.5.3 Range of floating-point types

In addition to the precision, a range can also be specified for a floating-point type. The syntax is similar to the one used for integer data types — using the **range** keyword. This simple example creates a new floating-point type based on the type **Float**, for a normalized range between **-1.0** and **1.0**:

Listing 10: floating_point_range.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Floating_Point_Range is
4   type T_Norm  is new Float range -1.0 .. 1.0;
5   A : T_Norm;
6 begin
7   A := 1.0;
8   Put_Line ("The value of A is "
9             & T_Norm'Image (A));
10 end Floating_Point_Range;
```

Runtime output

```
The value of A is 1.00000E+00
```

The application is responsible for ensuring that variables of this type stay within this range; otherwise an exception is raised. In this example, the exception **Constraint_Error** is raised when assigning **2.0** to the variable **A**:

Listing 11: floating_point_range_exception.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Floating_Point_Range_Exception is
4   type T_Norm  is new Float range -1.0 .. 1.0;
5   A : T_Norm;
6 begin
7   A := 2.0;
8   Put_Line ("The value of A is "
9             & T_Norm'Image (A));
10 end Floating_Point_Range_Exception;
```

Build output

```
floating_point_range_exception.adb:7:09: warning: value not in range of type "T_
->Norm" defined at line 4 [enabled by default]
floating_point_range_exception.adb:7:09: warning: Constraint_Error will be raised_
at run time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : floating_point_range_exception.adb:7 range check failed
```

Ranges can also be specified for custom floating-point types. For example:

Listing 12: custom_range_types.adb

```
1 with Ada.Text_Io;  use Ada.Text_Io;
2 with Ada.Numerics;  use Ada.Numerics;
3
4 procedure Custom_Range_Types is
5     type T6_Inv_Trig is
6         digits 6 range -Pi / 2.0 .. Pi / 2.0;
7 begin
8     null;
9 end Custom_Range_Types;
```

Build output

```
custom_range_types.adb:1:09: warning: no entities of "Ada.Text_Io" are referenced [-gnatwu]
custom_range_types.adb:1:20: warning: use clause for package "Text_Io" has no effect [-gnatwu]
custom_range_types.adb:5:09: warning: type "T6_Inv_Trig" is not referenced [-gnatwu]
```

In this example, we are defining a type called `T6_Inv_Trig`, which has a range from $-\pi / 2$ to $\pi / 2$ with a minimum precision of 6 digits. (π is defined in the predefined package `Ada.Numerics`.)

5.6 Strong typing

As noted earlier, Ada is strongly typed. As a result, different types of the same family are incompatible with each other; a value of one type cannot be assigned to a variable from the other type. For example:

Listing 13: illegal_example.adb

```
1 with Ada.Text_Io;  use Ada.Text_Io;
2
3 procedure Illegal_Example is
4     -- Declare two different floating point types
5     type Meters is new Float;
6     type Miles is new Float;
7
8     Dist_Imperial : Miles;
9
10    -- Declare a constant
11    Dist_Metric : constant Meters := 1000.0;
12 begin
13    -- Not correct: types mismatch
14    Dist_Imperial := Dist_Metric * 621.371e-6;
15    Put_Line (Miles'Image (Dist_Imperial));
16 end Illegal_Example;
```

Build output

```
illegal_example.adb:14:33: error: expected type "Miles" defined at line 6
illegal_example.adb:14:33: error: found type "Meters" defined at line 5
gprbuild: *** compilation phase failed
```

A consequence of these rules is that, in the general case, a "mixed mode" expression like `2 * 3.0` will trigger a compilation error. In a language like C or Python, such expressions are made valid by implicit conversions. In Ada, such conversions must be made explicit:

Listing 14: conv.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  procedure Conv is
3    type Meters is new Float;
4    type Miles is new Float;
5    Dist_Imperial : Miles;
6    Dist_Metric : constant Meters := 1000.0;
7  begin
8    Dist_Imperial := Miles (Dist_Metric) * 621.371e-6;
9    --                                     ^ Type conversion,
10   --                                     from Meters to Miles
11   -- Now the code is correct
12
13   Put_Line (Miles'Image (Dist_Imperial));
14 end Conv;
```

Runtime output

```
6.21371E-01
```

Of course, we probably do not want to write the conversion code every time we convert from meters to miles. The idiomatic Ada way in that case would be to introduce conversion functions along with the types.

Listing 15: conv.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Conv is
4    type Meters is new Float;
5    type Miles is new Float;
6
7    -- Function declaration, like procedure
8    -- but returns a value.
9    function To_Miles (M : Meters) return Miles is
10   --                                     ^ Return type
11  begin
12    return Miles (M) * 621.371e-6;
13  end To_Miles;
14
15  Dist_Imperial : Miles;
16  Dist_Metric   : constant Meters := 1000.0;
17  begin
18    Dist_Imperial := To_Miles (Dist_Metric);
19    Put_Line (Miles'Image (Dist_Imperial));
20  end Conv;
```

Runtime output

```
6.21371E-01
```

If you write a lot of numeric code, having to explicitly provide such conversions might seem painful at first. However, this approach brings some advantages. Notably, you can rely on

the absence of implicit conversions, which will in turn prevent some subtle errors.

In other languages

In C, for example, the rules for implicit conversions may not always be completely obvious. In Ada, however, the code will always do exactly what it seems to do. For example:

```
int a = 3, b = 2;
float f = a / b;
```

This code will compile fine, but the result of `f` will be 1.0 instead of 1.5, because the compiler will generate an integer division (three divided by two) that results in one. The software developer must be aware of data conversion issues and use an appropriate casting:

```
int a = 3, b = 2;
float f = (float)a / b;
```

In the corrected example, the compiler will convert both variables to their corresponding floating-point representation before performing the division. This will produce the expected result.

This example is very simple, and experienced C developers will probably notice and correct it before it creates bigger problems. However, in more complex applications where the type declaration is not always visible — e.g. when referring to elements of a `struct` — this situation might not always be evident and quickly lead to software defects that can be harder to find.

The Ada compiler, in contrast, will always reject code that mixes floating-point and integer variables without explicit conversion. The following Ada code, based on the erroneous example in C, will not compile:

Listing 16: main.adb

```
1 procedure Main is
2     A : Integer := 3;
3     B : Integer := 2;
4     F : Float;
5 begin
6     F := A / B;
7 end Main;
```

Build output

```
main.adb:6:04: warning: possibly useless assignment to "F", value might not bereferred [-gnatwm]
main.adb:6:11: error: expected type "Standard.Float"
main.adb:6:11: error: found type "Standard.Integer"
gprbuild: *** compilation phase failed
```

The offending line must be changed to `F := Float (A) / Float (B);` in order to be accepted by the compiler.

- You can use Ada's strong typing to help enforce invariants in your code, as in the example above: Since Miles and Meters are two different types, you cannot mistakenly convert an instance of one to an instance of the other.

5.7 Derived types

In Ada you can create new types based on existing ones. This is very useful: you get a type that has the same properties as some existing type but is treated as a distinct type in the interest of strong typing.

Listing 17: main.adb

```

1  procedure Main is
2      -- ID card number type,
3      -- incompatible with Integer.
4      type Social_Security_Number is new Integer
5          range 0 .. 999_99_999;
6          -- ^ Since a SSN has 9 digits
7          -- max., and cannot be
8          -- negative, we enforce
9          -- a validity constraint.
10
11     SSN : Social_Security_Number :=
12         555_55_5555;
13         -- ^ You can put underscores as
14         -- formatting in any number.
15
16     I : Integer;
17
18     -- The value -1 below will cause a
19     -- runtime error and a compile time
20     -- warning with GNAT.
21     Invalid : Social_Security_Number := -1;
22 begin
23     -- Illegal, they have different types:
24     I := SSN;
25
26     -- Likewise illegal:
27     SSN := I;
28
29     -- OK with explicit conversion:
30     I := Integer (SSN);
31
32     -- Likewise OK:
33     SSN := Social_Security_Number (I);
34 end Main;

```

Build output

```

main.adb:21:40: warning: value not in range of type "Social_Security_Number" ↴
    defined at line 4 [enabled by default]
main.adb:21:40: warning: Constraint_Error will be raised at run time [enabled by ↴
    default]
main.adb:24:09: error: expected type "Standard.Integer"
main.adb:24:09: error: found type "Social_Security_Number" defined at line 4
main.adb:27:11: error: expected type "Social_Security_Number" defined at line 4
main.adb:27:11: error: found type "Standard.Integer"
main.adb:33:04: warning: possibly useless assignment to "SSN", value might not be ↴
    referenced [-gnatwm]
gprbuild: *** compilation phase failed

```

The type `Social_Security` is said to be a *derived type*; its *parent type* is `Integer`.

As illustrated in this example, you can refine the valid range when defining a derived scalar type (such as integer, floating-point and enumeration).

The syntax for enumerations uses the `range <range>` syntax:

Listing 18: greet.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Greet is
4      type Days is (Monday, Tuesday, Wednesday,
5                     Thursday, Friday,
6                     Saturday, Sunday);
7
8      type Weekend_Days is new
9          Days range Saturday .. Sunday;
10         -- New type, where only Saturday and Sunday
11         -- are valid literals.
12 begin
13     null;
14 end Greet;

```

Build output

```

greet.adb:1:09: warning: no entities of "Ada.Text_Io" are referenced [-gnatwu]
greet.adb:1:19: warning: use clause for package "Text_Io" has no effect [-gnatwu]
greet.adb:4:18: warning: literal "Monday" is not referenced [-gnatwu]
greet.adb:4:26: warning: literal "Tuesday" is not referenced [-gnatwu]
greet.adb:4:35: warning: literal "Wednesday" is not referenced [-gnatwu]
greet.adb:5:18: warning: literal "Thursday" is not referenced [-gnatwu]
greet.adb:5:28: warning: literal "Friday" is not referenced [-gnatwu]
greet.adb:8:09: warning: type "Weekend_Days" is not referenced [-gnatwu]

```

5.8 Subtypes

As we are starting to see, types may be used in Ada to enforce constraints on the valid range of values. However, we sometimes want to enforce constraints on some values while staying within a single type. This is where subtypes come into play. A subtype does not introduce a new type.

Listing 19: greet.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Greet is
4      type Days is (Monday, Tuesday, Wednesday,
5                     Thursday, Friday,
6                     Saturday, Sunday);
7
8      -- Declaration of a subtype
9      subtype Weekend_Days is
10          Days range Saturday .. Sunday;
11          -- ^ Constraint of the subtype
12
13      M : Days := Sunday;
14
15      S : Weekend_Days := M;
16      -- No error here, Days and Weekend_Days
17      -- are of the same type.
18 begin
19     for I in Days loop
20         case I is
21             -- Just like a type, a subtype can

```

(continues on next page)

(continued from previous page)

```

22      -- be used as a range
23      when Weekend_Days =>
24          Put_Line ("Week end!");
25      when others =>
26          Put_Line ("Hello on "
27                      & Days'Image (I));
28      end case;
29  end loop;
30 end Greet;
```

Build output

```
greet.adb:13:04: warning: "M" is not modified, could be declared constant [-gnatwk]
greet.adb:15:04: warning: variable "S" is not referenced [-gnatwu]
```

Runtime output

```
Hello on MONDAY
Hello on TUESDAY
Hello on WEDNESDAY
Hello on THURSDAY
Hello on FRIDAY
Week end!
Week end!
```

Several subtypes are predefined in the standard package in Ada, and are automatically available to you:

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

While subtypes of a type are statically compatible with each other, constraints are enforced at run time: if you violate a subtype constraint, an exception will be raised.

Listing 20: greet.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Greet is
4      type Days is (Monday, Tuesday, Wednesday,
5                     Thursday, Friday,
6                     Saturday, Sunday);
7
8      subtype Weekend_Days is
9          Days range Saturday .. Sunday;
10
11     Day      : Days := Saturday;
12     Weekend : Weekend_Days;
13
14 begin
15     Weekend := Day;
16     --           ^ Correct: Same type, subtype
17     --           constraints are respected
18     Weekend := Monday;
19     --           ^ Wrong value for the subtype
20     --           Compiles, but exception at runtime
21 end Greet;
```

Build output

```
greet.adb:1:09: warning: no entities of "Ada.Text_Io" are referenced [-gnatwu]
greet.adb:1:19: warning: use clause for package "Text_Io" has no effect [-gnatwu]
```

(continues on next page)

(continued from previous page)

```

greet.adb:4:26: warning: literal "Tuesday" is not referenced [-gnatwu]
greet.adb:4:35: warning: literal "Wednesday" is not referenced [-gnatwu]
greet.adb:5:18: warning: literal "Thursday" is not referenced [-gnatwu]
greet.adb:5:28: warning: literal "Friday" is not referenced [-gnatwu]
greet.adb:11:04: warning: "Day" is not modified, could be declared constant [-gnatwk]
greet.adb:12:04: warning: variable "Weekend" is assigned but never read [-gnatwm]
greet.adb:14:04: warning: useless assignment to "Weekend", value overwritten at line 17 [-gnatwm]
greet.adb:17:04: warning: possibly useless assignment to "Weekend", value might not be referenced [-gnatwm]
greet.adb:17:15: warning: value not in range of type "Weekend_Days" defined at line 8 [enabled by default]
greet.adb:17:15: warning: Constraint_Error will be raised at run time [enabled by default]

```

Runtime output

```
raised CONSTRAINT_ERROR : greet.adb:17 range check failed
```

5.8.1 Subtypes as type aliases

Previously, we've seen that we can create new types by declaring `type Miles is new Float`. We could also create type aliases, which generate alternative names — *aliases* — for known types. Note that type aliases are sometimes called *type synonyms*.

We achieve this in Ada by using subtypes without new constraints. In this case, however, we don't get all of the benefits of Ada's strong type checking. Let's rewrite an example using type aliases:

Listing 21: undetected_imperial_metric_error.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Undetected_Imperial_Metric_Error is
4      -- Declare two type aliases
5      subtype Meters is Float;
6      subtype Miles is Float;
7
8      Dist_Imperial : Miles;
9
10     -- Declare a constant
11     Dist_Metric : constant Meters := 100.0;
12 begin
13     -- No conversion to Miles type required:
14     Dist_Imperial := (Dist_Metric * 1609.0) / 1000.0;
15
16     -- Not correct, but undetected:
17     Dist_Imperial := Dist_Metric;
18
19     Put_Line (Miles'Image (Dist_Imperial));
20 end Undetected_Imperial_Metric_Error;

```

Build output

```
undetected_imperial_metric_error.adb:14:04: warning: useless assignment to "Dist_Imperial", value overwritten at line 17 [-gnatwm]
```

Runtime output

```
1.00000E+02
```

In the example above, the fact that both Meters and Miles are subtypes of **Float** allows us to mix variables of both types without type conversion. This, however, can lead to all sorts of programming mistakes that we'd like to avoid, as we can see in the undetected error highlighted in the code above. In that example, the error in the assignment of a value in meters to a variable meant to store values in miles remains undetected because both Meters and Miles are subtypes of **Float**. Therefore, the recommendation is to use strong typing — via **type X is new Y** — for cases such as the one above.

There are, however, many situations where type aliases are useful. For example, in an application that uses floating-point types in multiple contexts, we could use type aliases to indicate additional meaning to the types or to avoid long variable names. For example, instead of writing:

```
Paid_Amount, Due_Amount : Float;
```

We could write:

```
subtype Amount is Float;
```

```
Paid, Due : Amount;
```

In other languages

In C, for example, we can use a **typedef** declaration to create a type alias. For example:

```
typedef float meters;
```

This corresponds to the declaration that we've seen above using subtypes. Other programming languages include this concept in similar ways. For example:

- C++: `using meters = float;`
 - Swift: `typealias Meters = Double`
 - Kotlin: `typealias Meters = Double`
 - Haskell: `type Meters = Float`
-

Note, however, that subtypes in Ada correspond to type aliases if, and only if, they don't have new constraints. Thus, if we add a new constraint to a subtype declaration, we don't have a type alias anymore. For example, the following declaration *can't* be considered a type alias of **Float**:

```
subtype Meters is Float range 0.0 .. 1_000_000.0;
```

Let's look at another example:

```
subtype Degree_Celsius is Float;  
  
subtype Liquid_Water_Temperature is  
    Degree_Celsius range 0.0 .. 100.0;  
  
subtype Running_Water_Temperature is  
    Liquid_Water_Temperature;
```

In this example, `Liquid_Water_Temperature` isn't an alias of `Degree_Celsius`, since it adds a new constraint that wasn't part of the declaration of the `Degree_Celsius`. However, we do have two type aliases here:

- `Degree_Celsius` is an alias of **Float**;

- `Running_Water_Temperature` is an alias of `Liquid_Water_Temperature`, even if `Liquid_Water_Temperature` itself has a constrained range.

RECORDS

So far, all the types we have encountered have values that are not decomposable: each instance represents a single piece of data. Now we are going to see our first class of composite types: records.

Records allow composing a value out of instances of other types. Each of those instances will be given a name. The pair consisting of a name and an instance of a specific type is called a field, or a component.

6.1 Record type declaration

Here is an example of a simple record declaration:

```
type Date is record
    -- The following declarations are
    -- components of the record
    Day : Integer range 1 .. 31;
    Month : Months;
    -- You can add custom constraints
    -- on fields
    Year : Integer range 1 .. 3000;
end record;
```

Fields look a lot like variable declarations, except that they are inside of a record definition. And as with variable declarations, you can specify additional constraints when supplying the subtype of the field.

```
type Date is record
    Day : Integer range 1 .. 31;
    Month : Months := January;
    -- This component has a default value
    Year : Integer range 1 .. 3000 := 2012;
    --                                         ^ Default value
end record;
```

Record components can have default values. When a variable having the record type is declared, a field with a default initialization will be automatically set to this value. The value can be any expression of the component type, and may be run-time computable.

In the remaining sections of this chapter, we see how to use record types. In addition to that, we discuss more about records in [another chapter](#) (page 95).

6.2 Aggregates

```

Ada_Birthday    : Date := (10, December, 1815);
Leap_Day_2020   : Date := (Day    => 29,
                           Month  => February,
                           Year   => 2020);
--                                         ^ By name

```

Records have a convenient notation for expressing values, illustrated above. This notation is called aggregate notation, and the literals are called aggregates. They can be used in a variety of contexts that we will see throughout the course, one of which is to initialize records.

An aggregate is a list of values separated by commas and enclosed in parentheses. It is allowed in any context where a value of the record is expected.

Values for the components can be specified positionally, as in Ada_Birthday example, or by name, as in Leap_Day_2020. A mixture of positional and named values is permitted, but you cannot use a positional notation after a named one.

6.3 Component selection

To access components of a record instance, you use an operation that is called component selection. This is achieved by using the dot notation. For example, if we declare a variable Some_Day of the Date record type mentioned above, we can access the Year component by writing Some_Day.Year.

Let's look at an example:

Listing 1: record_selection.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Record_Selection is
4
5      type Months is
6          (January, February, March, April,
7           May, June, July, August, September,
8           October, November, December);
9
10     type Date is record
11         Day   : Integer range 1 .. 31;
12         Month : Months;
13         Year  : Integer range 1 .. 3000 := 2032;
14     end record;
15
16     procedure Display_Date (D : Date) is
17 begin
18     Put_Line ("Day:" & Integer'Image (D.Day)
19                 & ", Month: "
20                 & Months'Image (D.Month)
21                 & ", Year:"
22                 & Integer'Image (D.Year));
23 end Display_Date;
24
25     Some_Day : Date := (1, January, 2000);
26
27 begin
28     Display_Date (Some_Day);

```

(continues on next page)

(continued from previous page)

```

29      Put_Line ("Changing year...");  

30      Some_Day.Year := 2001;  

31  

32      Display_Date (Some_Day);  

33  end Record_Selection;

```

Runtime output

```

Day: 1, Month: JANUARY, Year: 2000  

Changing year...  

Day: 1, Month: JANUARY, Year: 2001

```

As you can see in this example, we can use the dot notation in the expression D.Year or Some_Day.Year to access the information stored in that component, as well as to modify this information in assignments. To be more specific, when we use D.Year in the call to Put_Line, we're retrieving the information stored in that component. When we write Some_Day.Year := 2001, we're overwriting the information that was previously stored in the Year component of Some_Day.

6.4 Renaming

In previous chapters, we've discussed *subprogram* (page 29) and *package* (page 40) renaming. We can rename record components as well. Instead of writing the full component selection using the dot notation, we can declare an alias that allows us to access the same component. This is useful to simplify the implementation of a subprogram, for example.

We can rename record components by using the **renames** keyword in a variable declaration. For example:

```
Some_Day : Date;  
Y          : Integer renames Some_Day.Year;
```

Here, Y is an alias, so that every time we use Y, we are really using the Year component of Some_Day.

Let's look at a complete example:

Listing 2: dates.ads

```

1 package Dates is  

2  

3     type Months is  

4         (January, February, March, April,  

5          May, June, July, August, September,  

6          October, November, December);  

7  

8     type Date is record  

9         Day   : Integer range 1 .. 31;  

10        Month : Months;  

11        Year  : Integer range 1 .. 3000 := 2032;  

12    end record;  

13  

14    procedure Increase_Month (Some_Day : in out Date);  

15  

16    procedure Display_Month (Some_Day : Date);  

17  

18 end Dates;

```

Listing 3: dates.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  package body Dates is
4
5      procedure Increase_Month (Some_Day : in out Date) is
6          -- Renaming components from
7          -- the Date record
8          M : Months renames Some_Day.Month;
9          Y : Integer renames Some_Day.Year;
10
11         -- Renaming function (for Months
12         -- enumeration)
13         function Next (M : Months) return Months
14             renames Months'Succ;
15     begin
16         if M = December then
17             M := January;
18             Y := Y + 1;
19         else
20             M := Next (M);
21         end if;
22     end Increase_Month;
23
24     procedure Display_Month (Some_Day : Date) is
25         -- Renaming components from
26         -- the Date record
27         M : Months renames Some_Day.Month;
28         Y : Integer renames Some_Day.Year;
29     begin
30         Put_Line ("Month: "
31                     & Months'Image (M)
32                     & ", Year:"
33                     & Integer'Image (Y));
34     end Display_Month;
35
36 end Dates;

```

Listing 4: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2  with Dates;      use Dates;
3
4  procedure Main is
5      D : Date := (1, January, 2000);
6  begin
7      Display_Month (D);
8
9      Put_Line ("Increasing month...");
10     Increase_Month (D);
11
12     Display_Month (D);
13 end Main;

```

Runtime output

```

Month: JANUARY, Year: 2000
Increasing month...
Month: FEBRUARY, Year: 2000

```

We apply renaming to two components of the Date record in the implementation of the In-

crease_Month procedure. Then, instead of directly using Some_Day.Month and Some_Day.Year in the next operations, we simply use the renamed versions M and Y.

Note that, in the example above, we also rename Months' `Succ` — which is the function that gives us the next month — to `Next`.

ARRAYS

Arrays provide another fundamental family of composite types in Ada.

7.1 Array type declaration

Arrays in Ada are used to define contiguous collections of elements that can be selected by indexing. Here's a simple example:

Listing 1: greet.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Greet is
4      type My_Int is range 0 .. 1000;
5      type Index is range 1 .. 5;
6
7      type My_Int_Array is
8          array (Index) of My_Int;
9          --           ^ Type of elements
10         --        ^ Bounds of the array
11     Arr : My_Int_Array := (2, 3, 5, 7, 11);
12         --           ^ Array literal
13         --                   (aggregate)
14
15     V : My_Int;
16 begin
17     for I in Index loop
18         V := Arr (I);
19         --           ^ Take the Ith element
20         Put (My_Int'Image (V));
21     end loop;
22     New_Line;
23 end Greet;
```

Build output

```
greet.adb:11:04: warning: "Arr" is not modified, could be declared constant [-
→gnatwk]
```

Runtime output

```
2 3 5 7 11
```

The first point to note is that we specify the index type for the array, rather than its size. Here we declared an integer type named `Index` ranging from `1` to `5`, so each array instance will have 5 elements, with the initial element at index 1 and the last element at index 5.

Although this example used an integer type for the index, Ada is more general: any discrete type is permitted to index an array, including [Enum types](#) (page 47). We will soon see what that means.

Another point to note is that querying an element of the array at a given index uses the same syntax as for function calls: that is, the array object followed by the index in parentheses.

Thus when you see an expression such as `A (B)`, whether it is a function call or an array subscript depends on what `A` refers to.

Finally, notice how we initialize the array with the `(2, 3, 5, 7, 11)` expression. This is another kind of aggregate in Ada, and is in a sense a literal expression for an array, in the same way that `3` is a literal expression for an integer. The notation is very powerful, with a number of properties that we will introduce later. A detailed overview appears in the notation of [aggregate types](#) (page 83).

Unrelated to arrays, the example also illustrated two procedures from `Ada.Text_Io`:

- `Put`, which displays a string without a terminating end of line
- `New_Line`, which outputs an end of line

Let's now delve into what it means to be able to use any discrete type to index into the array.

In other languages

Semantically, an array object in Ada is the entire data structure, and not simply a handle or pointer. Unlike C and C++, there is no implicit equivalence between an array and a pointer to its initial element.

Listing 2: `array_bounds_example.adb`

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Array_Bounds_Example is
4      type My_Int is range 0 .. 1000;
5      type Index is range 11 .. 15;
6          --                                         ^ Low bound can be any value
7      type My_Int_Array is array (Index) of My_Int;
8      Tab : constant My_Int_Array := (2, 3, 5, 7, 11);
9  begin
10     for I in Index loop
11         Put (My_Int'Image (Tab (I)));
12     end loop;
13     New_Line;
14 end Array_Bounds_Example;
```

Runtime output

```
2 3 5 7 11
```

One effect is that the bounds of an array can be any values. In the first example we constructed an array type whose first index is `1`, but in the example above we declare an array type whose first index is `11`.

That's perfectly fine in Ada, and moreover since we use the index type as a range to iterate over the array indices, the code using the array does not need to change.

That leads us to an important consequence with regard to code dealing with arrays. Since the bounds can vary, you should not assume / hard-code specific bounds when iterating / using arrays. That means the code above is good, because it uses the index type, but a for loop as shown below is bad practice even though it works correctly:

```

for I in 11 .. 15 loop
    Tab (I) := Tab (I) * 2;
end loop;

```

Since you can use any discrete type to index an array, enumeration types are permitted.

Listing 3: month_example.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Month_Example is
4      type Month_Duration is range 1 .. 31;
5      type Month is (Jan, Feb, Mar, Apr,
6                      May, Jun, Jul, Aug,
7                      Sep, Oct, Nov, Dec);
8
9      type My_Int_Array is
10         array (Month) of Month_Duration;
11         --           ^ Can use an enumeration type
12         --           as the index
13
14      Tab : constant My_Int_Array :=
15          --           ^ constant is like a variable but
16          --           cannot be modified
17          (31, 28, 31, 30, 31, 30,
18             31, 31, 30, 31, 30, 31);
19          -- Maps months to number of days
20          -- (ignoring leap years)
21
22      Feb_Days : Month_Duration := Tab (Feb);
23      -- Number of days in February
24 begin
25     for M in Month loop
26         Put_Line
27             (Month'Image (M) & " has "
28              & Month_Duration'Image (Tab (M))
29              & " days.");
30             --           ^ Concatenation operator
31     end loop;
32 end Month_Example;

```

Build output

```
month_example.adb:22:04: warning: variable "Feb_Days" is not referenced [-gnatwu]
```

Runtime output

```

JAN has 31 days.
FEB has 28 days.
MAR has 31 days.
APR has 30 days.
MAY has 31 days.
JUN has 30 days.
JUL has 31 days.
AUG has 31 days.
SEP has 30 days.
OCT has 31 days.
NOV has 30 days.
DEC has 31 days.

```

In the example above, we are:

- Creating an array type mapping months to month durations in days.

- Creating an array, and instantiating it with an aggregate mapping months to their actual durations in days.
- Iterating over the array, printing out the months, and the number of days for each.

Being able to use enumeration values as indices is very helpful in creating mappings such as shown above one, and is an often used feature in Ada.

7.2 Indexing

We have already seen the syntax for selecting elements of an array. There are however a few more points to note.

First, as is true in general in Ada, the indexing operation is strongly typed. If you use a value of the wrong type to index the array, you will get a compile-time error.

Listing 4: greet.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4      type My_Int is range 0 .. 1000;
5
6      type My_Index    is range 1 .. 5;
7      type Your_Index is range 1 .. 5;
8
9      type My_Int_Array is array (My_Index) of My_Int;
10     Tab : My_Int_Array := (2, 3, 5, 7, 11);
11
12 begin
13     for I in Your_Index loop
14         Put (My_Int'Image (Tab (I)));
15         --                                         ^ Compile time error
16     end loop;
17     New_Line;
18 end Greet;
```

Build output

```

greet.adb:13:31: error: expected type "My_Index" defined at line 6
greet.adb:13:31: error: found type "Your_Index" defined at line 7
gprbuild: *** compilation phase failed
```

Second, arrays in Ada are bounds checked. This means that if you try to access an element outside of the bounds of the array, you will get a run-time error instead of accessing random memory as in unsafe languages.

Listing 5: greet.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4      type My_Int is range 0 .. 1000;
5      type Index is range 1 .. 5;
6      type My_Int_Array is array (Index) of My_Int;
7      Tab : My_Int_Array := (2, 3, 5, 7, 11);
8
9 begin
10     for I in Index range 2 .. 6 loop
11         Put (My_Int'Image (Tab (I)));
12         --                                         ^ Will raise an
13         --                                         exception when
```

(continues on next page)

(continued from previous page)

```

13      --          I = 6
14  end loop;
15  New_Line;
16 end Greet;
```

Build output

```

greet.adb:7:04: warning: "Tab" is not modified, could be declared constant [-
→gnatwk]
greet.adb:9:30: warning: static value out of range of type "Index" defined at line ↵
→5 [enabled by default]
greet.adb:9:30: warning: Constraint_Error will be raised at run time [enabled by ↵
→default]
greet.adb:9:30: warning: suspicious loop bound out of range of loop subtype ↵
→[enabled by default]
greet.adb:9:30: warning: loop executes zero times or raises Constraint_Error ↵
→[enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : greet.adb:9 range check failed
```

7.3 Simpler array declarations

In the previous examples, we have always explicitly created an index type for the array. While this can be useful for typing and readability purposes, sometimes you simply want to express a range of values. Ada allows you to do that, too.

Listing 6: simple_array_bounds.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Simple_Array_Bounds is
4   type My_Int is range 0 .. 1000;
5   type My_Int_Array is array (1 .. 5) of My_Int;
6   --           ^ Subtype of Integer
7   Tab : constant My_Int_Array := (2, 3, 5, 7, 11);
8 begin
9   for I in 1 .. 5 loop
10    --           ^ Subtype of Integer
11    Put (My_Int'Image (Tab (I)));
12  end loop;
13  New_Line;
14 end Simple_Array_Bounds;
```

Runtime output

```
2 3 5 7 11
```

This example defines the range of the array via the range syntax, which specifies an anonymous subtype of `Integer` and uses it to index the array.

This means that the type of the index is `Integer`. Similarly, when you use an anonymous range in a for loop as in the example above, the type of the iteration variable is also `Integer`, so you can use `I` to index `Tab`.

You can also use a named subtype for the bounds for an array.

7.4 Range attribute

We noted earlier that hard coding bounds when iterating over an array is a bad idea, and showed how to use the array's index type/subtype to iterate over its range in a for loop. That raises the question of how to write an iteration when the array has an anonymous range for its bounds, since there is no name to refer to the range. Ada solves that via several attributes of array objects:

Listing 7: range_example.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Range_Example is
4      type My_Int is range 0 .. 1000;
5      type My_Int_Array is array (1 .. 5) of My_Int;
6      Tab : constant My_Int_Array := (2, 3, 5, 7, 11);
7  begin
8      for I in Tab'Range loop
9          --           ^ Gets the range of Tab
10         Put (My_Int'Image (Tab (I)));
11     end loop;
12     New_Line;
13 end Range_Example;

```

Runtime output

```
2 3 5 7 11
```

If you want more fine grained control, you can use the separate attributes '`First`' and '`Last`'.

Listing 8: array_attributes_example.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Array_Attributes_Example is
4      type My_Int is range 0 .. 1000;
5      type My_Int_Array is array (1 .. 5) of My_Int;
6      Tab : My_Int_Array := (2, 3, 5, 7, 11);
7  begin
8      for I in Tab'First .. Tab'Last - 1 loop
9          --           ^ Iterate on every index
10         --           except the last
11         Put (My_Int'Image (Tab (I)));
12     end loop;
13     New_Line;
14 end Array_Attributes_Example;

```

Build output

```
array_attributes_example.adb:6:04: warning: "Tab" is not modified, could be_u
→declared constant [-gnatwk]
```

Runtime output

```
2 3 5 7
```

The '`Range`', '`First`' and '`Last`' attributes in these examples could also have been applied to the array type name, and not just the array instances.

Although not illustrated in the above examples, another useful attribute for an array instance A is A'`Length`', which is the number of elements that A contains.

It is legal and sometimes useful to have a "null array", which contains no elements. To get this effect, define an index range whose upper bound is less than the lower bound.

7.5 Unconstrained arrays

Let's now consider one of the most powerful aspects of Ada's array facility.

Every array type we have defined so far has a fixed size: every instance of this type will have the same bounds and therefore the same number of elements and the same size.

However, Ada also allows you to declare array types whose bounds are not fixed: in that case, the bounds will need to be provided when creating instances of the type.

Listing 9: unconstrained_array_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Unconstrained_Array_Example is
4      type Days is (Monday, Tuesday, Wednesday,
5                     Thursday, Friday,
6                     Saturday, Sunday);
7
8      type Workload_Type is
9          array (Days range <>) of Natural;
10         -- Indefinite array type
11         -- ^ Bounds are of type Days,
12         -- but not known
13
14      Workload : constant
15          Workload_Type (Monday .. Friday) := 
16              --           ^ Specify the bounds
17              --           when declaring
18              (Friday => 7, others => 8);
19              --           ^ Default value
20              -- ^ Specify element by name of index
21 begin
22     for I in Workload'Range loop
23         Put_Line (Integer'Image (Workload (I)));
24     end loop;
25 end Unconstrained_Array_Example;

```

Build output

```

unconstrained_array_example.adb:4:26: warning: literal "Tuesday" is not referenced ↵ [-gnatwu]
unconstrained_array_example.adb:4:35: warning: literal "Wednesday" is not ↵ referenced [-gnatwu]
unconstrained_array_example.adb:5:18: warning: literal "Thursday" is not ↵ referenced [-gnatwu]
unconstrained_array_example.adb:6:18: warning: literal "Saturday" is not ↵ referenced [-gnatwu]
unconstrained_array_example.adb:6:28: warning: literal "Sunday" is not referenced ↵ [-gnatwu]

```

Runtime output

```

8
8
8
8
7

```

The fact that the bounds of the array are not known is indicated by the Days **range** \leftrightarrow syntax. Given a discrete type Discrete_Type, if we use Discrete_Type for the index in an array type then Discrete_Type serves as the type of the index and comprises the range of index values for each array instance.

If we define the index as Discrete_Type **range** \leftrightarrow then Discrete_Type serves as the type of the index, but different array instances may have different bounds from this type

An array type that is defined with the Discrete_Type **range** \leftrightarrow syntax for its index is referred to as an unconstrained array type, and, as illustrated above, the bounds need to be provided when an instance is created.

The above example also shows other forms of the aggregate syntax. You can specify associations by name, by giving the value of the index on the left side of an arrow association. **1 => 2** thus means "assign value 2 to the element at index 1 in my array". **others => 8** means "assign value 8 to every element that wasn't previously assigned in this aggregate".

Attention: The so-called "box" notation (\leftrightarrow) is commonly used as a wildcard or placeholder in Ada. You will often see it when the meaning is "what is expected here can be anything".

In other languages

While unconstrained arrays in Ada might seem similar to variable length arrays in C, they are in reality much more powerful, because they're truly first-class values in the language. You can pass them as parameters to subprograms or return them from functions, and they implicitly contain their bounds as part of their value. This means that it is useless to pass the bounds or length of an array explicitly along with the array, because they are accessible via the '**First**', '**Last**', '**Range**' and '**Length**' attributes explained earlier.

Although different instances of the same unconstrained array type can have different bounds, a specific instance has the same bounds throughout its lifetime. This allows Ada to implement unconstrained arrays efficiently; instances can be stored on the stack and do not require heap allocation as in languages like Java.

7.6 Predefined array type: String

A recurring theme in our introduction to Ada types has been the way important built-in types like **Boolean** or **Integer** are defined through the same facilities that are available to the user. This is also true for strings: The **String** type in Ada is a simple array.

Here is how the string type is defined in Ada:

```
type String is array (Positive range <>) of Character;
```

The only built-in feature Ada adds to make strings more ergonomic is custom literals, as we can see in the example below.

Hint: String literals are a syntactic sugar for aggregates, so that in the following example, A and B have the same value.

Listing 10: string_literals.ads

```
1 package String_Literals is
2   -- Those two declarations are equivalent
```

(continues on next page)

(continued from previous page)

```

3   A : String (1 .. 11) := "Hello World";
4   B : String (1 .. 11) :=
5     ('H', 'e', 'l', 'l', 'o', ' ', 
6      'W', 'o', 'r', 'l', 'd');
7 end String_Literals;

```

Listing 11: greet.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   Message : String (1 .. 11) := "dlroW olleH";
5   --           ^ Pre-defined array type.
6   --           Component type is Character
7 begin
8   for I in reverse Message'Range loop
9     --           ^ Iterate in reverse order
10    Put (Message (I));
11   end loop;
12   New_Line;
13 end Greet;

```

However, specifying the bounds of the object explicitly is a bit of a hassle; you have to manually count the number of characters in the literal. Fortunately, Ada gives you an easier way.

You can omit the bounds when creating an instance of an unconstrained array type if you supply an initialization, since the bounds can be deduced from the initialization expression.

Listing 12: greet.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   Message : constant String := "dlroW olleH";
5   --           ^ Bounds are automatically
6   --           computed from
7   --           initialization value
8 begin
9   for I in reverse Message'Range loop
10    Put (Message (I));
11   end loop;
12   New_Line;
13 end Greet;

```

Runtime output

```
Hello World
```

Listing 13: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Integer_Array is array (Natural range <>) of Integer;
5
6   My_Array : constant Integer_Array := (1, 2, 3, 4);
7   --           ^ Bounds are automatically
8   --           computed from

```

(continues on next page)

(continued from previous page)

```

9      --           initialization value
10     begin
11       null;
12   end Main;
```

Attention: As you can see above, the standard **String** type in Ada is an array. As such, it shares the advantages and drawbacks of arrays: a **String** value is stack allocated, it is accessed efficiently, and its bounds are immutable.

If you want something akin to C++'s `std::string`, you can use *Unbounded Strings* (page 232) from Ada's standard library. This type is more like a mutable, automatically managed string buffer to which you can add content.

7.7 Restrictions

A very important point about arrays: bounds *have* to be known when instances are created. It is for example illegal to do the following.

```

declare
  A : String;
begin
  A := "World";
end;
```

Also, while you of course can change the values of elements in an array, you cannot change the array's bounds (and therefore its size) after it has been initialized. So this is also illegal:

```

declare
  A : String := "Hello";
begin
  A := "World";      -- OK: Same size
  A := "Hello World"; -- Not OK: Different size
end;
```

Also, while you can expect a warning for this kind of error in very simple cases like this one, it is impossible for a compiler to know in the general case if you are assigning a value of the correct length, so this violation will generally result in a run-time error.

Attention

While we will learn more about this later, it is important to know that arrays are not the only types whose instances might be of unknown size at compile-time.

Such objects are said to be of an *indefinite subtype*, which means that the subtype size is not known at compile time, but is dynamically computed (at run time).

Listing 14: indefinite_subtypes.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Indefinite_Subtypes is
4    function Get_Number return Integer is
5    begin
6      return Integer'Value (Get_Line);
7    end Get_Number;
```

(continues on next page)

(continued from previous page)

```

8   A : String := "Hello";
9   -- Indefinite subtype
10
11  B : String (1 .. 5) := "Hello";
12  -- Definite subtype
13
14  C : String (1 .. Get_Number);
15  -- Indefinite subtype
16  -- (Get_Number's value is computed at
17  -- run-time)
18
19 begin
20   null;
21 end Indefinite_Subtypes;

```

Here, the '`Value`' attribute converts the string to an integer.

7.8 Returning unconstrained arrays

The return type of a function can be any type; a function can return a value whose size is unknown at compile time. Likewise, the parameters can be of any type.

For example, this is a function that returns an unconstrained `String`:

Listing 15: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5    type Days is (Monday, Tuesday, Wednesday,
6                  Thursday, Friday,
7                  Saturday, Sunday);
8
9    function Get_Day_Name (Day : Days := Monday)
10                      return String is
11
12    begin
13      return
14        (case Day is
15          when Monday    => "Monday",
16          when Tuesday   => "Tuesday",
17          when Wednesday => "Wednesday",
18          when Thursday  => "Thursday",
19          when Friday   => "Friday",
20          when Saturday  => "Saturday",
21          when Sunday   => "Sunday");
22
23  begin
24    Put_Line ("First day is "
25              & Get_Day_Name (Days'First));
26 end Main;

```

Runtime output

```
First day is Monday
```

(This example is for illustrative purposes only. There is a built-in mechanism, the '`Image`

attribute for scalar types, that returns the name (as a **String**) of any element of an enumeration type. For example Days' **Image**(Monday) is "MONDAY".)

In other languages

Returning variable size objects in languages lacking a garbage collector is a bit complicated implementation-wise, which is why C and C++ don't allow it, preferring to depend on explicit dynamic allocation / free from the user.

The problem is that explicit storage management is unsafe as soon as you want to collect unused memory. Ada's ability to return variable size objects will remove one use case for dynamic allocation, and hence, remove one potential source of bugs from your programs.

Rust follows the C/C++ model, but with safe pointer semantics. However, dynamic allocation is still used. Ada can benefit from a possible performance edge because it can use any model.

7.9 Declaring arrays (2)

While we can have array types whose size and bounds are determined at run time, the array's component type needs to be of a definite and constrained type.

Thus, if you need to declare, for example, an array of strings, the **String** subtype used as component will need to have a fixed size.

Listing 16: show_days.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Days is
4      type Days is (Monday, Tuesday, Wednesday,
5                     Thursday, Friday,
6                     Saturday, Sunday);
7
8      subtype Day_Name is String (1 .. 2);
9      -- Subtype of string with known size
10
11     type Days_Name_Type is
12         array (Days) of Day_Name;
13         -- ^ Type of the index
14         --          ^ Type of the element.
15         --          Must be definite
16
17     Names : constant Days_Name_Type :=
18         ("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su");
19         -- Initial value given by aggregate
20 begin
21     for I in Names'Range loop
22         Put_Line (Names (I));
23     end loop;
24 end Show_Days;
```

Build output

```
show_days.adb:4:18: warning: literal "Monday" is not referenced [-gnatwu]
show_days.adb:4:26: warning: literal "Tuesday" is not referenced [-gnatwu]
show_days.adb:4:35: warning: literal "Wednesday" is not referenced [-gnatwu]
show_days.adb:5:18: warning: literal "Thursday" is not referenced [-gnatwu]
```

(continues on next page)

(continued from previous page)

```
show_days.adb:5:28: warning: literal "Friday" is not referenced [-gnatwu]
show_days.adb:6:18: warning: literal "Saturday" is not referenced [-gnatwu]
show_days.adb:6:28: warning: literal "Sunday" is not referenced [-gnatwu]
```

Runtime output

```
Mo
Tu
We
Th
Fr
Sa
Su
```

7.10 Array slices

One last feature of Ada arrays that we're going to cover is array slices. It is possible to take and use a slice of an array (a contiguous sequence of elements) as a name or a value.

Listing 17: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4    Buf : String := "Hello ...";
5
6    Full_Name : String := "John Smith";
7  begin
8    Buf (7 .. 9) := "Bob";
9    -- Careful! This works because the string
10   -- on the right side is the same length as
11   -- the replaced slice!
12
13   -- Prints "Hello Bob"
14   Put_Line (Buf);
15
16   -- Prints "Hi John"
17   Put_Line ("Hi " & Full_Name (1 .. 4));
18 end Main;
```

Build output

```
main.adb:6:05: warning: "Full_Name" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Hello Bob
Hi John
```

As we can see above, you can use a slice on the left side of an assignment, to replace only part of an array.

A slice of an array is of the same type as the array, but has a different subtype, constrained by the bounds of the slice.

Attention: Ada has [multidimensional arrays¹⁰](#), which are not covered in this course. Slices will only work on one dimensional arrays.

7.11 Renaming

So far, we've seen that the following elements can be renamed: [subprograms](#) (page 29), [packages](#) (page 40), and [record components](#) (page 63). We can also rename objects by using the `renames` keyword. This allows for creating alternative names for these objects. Let's look at an example:

Listing 18: measurements.ads

```
1 package Measurements is
2
3     subtype Degree_Celsius is Float;
4
5     Current_Temperature : Degree_Celsius;
6
7 end Measurements;
```

Listing 19: main.adb

```
1 with Ada.Text_Io;  use Ada.Text_Io;
2 with Measurements;
3
4 procedure Main is
5     subtype Degrees is Measurements.Degree_Celsius;
6
7     T : Degrees
8         renames Measurements.Current_Temperature;
9 begin
10    T := 5.0;
11
12    Put_Line (Degrees'Image (T));
13    Put_Line (Degrees'Image
14              (Measurements.Current_Temperature));
15
16    T := T + 2.5;
17
18    Put_Line (Degrees'Image (T));
19    Put_Line (Degrees'Image
20              (Measurements.Current_Temperature));
21 end Main;
```

Runtime output

```
5.00000E+00
5.00000E+00
7.50000E+00
7.50000E+00
```

In the example above, we declare a variable `T` by renaming the `Current_Temperature` object from the `Measurements` package. As you can see by running this example, both `Current_Temperature` and its alternative name `T` have the same values:

- first, they show the value 5.0

¹⁰ <http://www.adu-auth.org/standards/12rm/html/RM-3-6.html>

- after the addition, they show the value 7.5.

This is because they are essentially referring to the same object, but with two different names.

Note that, in the example above, we're using Degrees as an alias of Degree_Celsius. We discussed this method *earlier in the course* (page 57).

Renaming can be useful for improving the readability of more complicated array indexing. Instead of explicitly using indices every time we're accessing certain positions of the array, we can create shorter names for these positions by renaming them. Let's look at the following example:

Listing 20: colors.ads

```

1 package Colors is
2
3   type Color is (Black, Red, Green, Blue, White);
4
5   type Color_Array is
6     array (Positive range <>) of Color;
7
8   procedure Reverse_It (X : in out Color_Array);
9
10 end Colors;
```

Listing 21: colors.adb

```

1 package body Colors is
2
3   procedure Reverse_It (X : in out Color_Array) is
4   begin
5     for I in X'First .. (X'Last + X'First) / 2 loop
6       declare
7         Tmp      : Color;
8         X_Left  : Color
9           renames X (I);
10        X_Right : Color
11          renames X (X'Last + X'First - I);
12      begin
13        Tmp      := X_Left;
14        X_Left  := X_Right;
15        X_Right := Tmp;
16      end;
17    end loop;
18  end Reverse_It;
19
20 end Colors;
```

Listing 22: test_reverse_colors.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2
3 with Colors; use Colors;
4
5 procedure Test_Reverse_Colors is
6
7   My_Colors : Color_Array (1 .. 5) :=
8     (Black, Red, Green, Blue, White);
9
10 begin
11   for C of My_Colors loop
```

(continues on next page)

(continued from previous page)

```
12     Put_Line ("My_Color: " & Color'Image (C));
13 end loop;
14
15 New_Line;
16 Put_Line ("Reversing My_Color...");
17 New_Line;
18 Reverse_It (My_Colors);
19
20 for C of My_Colors loop
21     Put_Line ("My_Color: " & Color'Image (C));
22 end loop;
23
24 end Test_Reverse_Colors;
```

Runtime output

```
My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK
```

In the example above, package Colors implements the procedure Reverse_It by declaring new names for two positions of the array. The actual implementation becomes easy to read:

```
begin
  Tmp      := X_Left;
  X_Left   := X_Right;
  X_Right := Tmp;
end;
```

Compare this to the alternative version without renaming:

```
begin
  Tmp          := X (I);
  X (I)        := X (X'Last + X'First - I);
  X (X'Last + X'First - I) := Tmp;
end;
```

MORE ABOUT TYPES

8.1 Aggregates: A primer

So far, we have talked about aggregates quite a bit and have seen a number of examples. Now we will revisit this feature in some more detail.

An Ada aggregate is, in effect, a literal value for a composite type. It's a very powerful notation that helps you to avoid writing procedural code for the initialization of your data structures in many cases.

A basic rule when writing aggregates is that *every component* of the array or record has to be specified, even components that have a default value.

This means that the following code is incorrect:

Listing 1: incorrect.ads

```
1 package Incorrect is
2     type Point is record
3         X, Y : Integer := 0;
4     end record;
5
6     Origin : Point := (X => 0);
7 end Incorrect;
```

Build output

```
incorrect.ads:6:22: error: no value supplied for component "Y"
gprbuild: *** compilation phase failed
```

There are a few shortcuts that you can use to make the notation more convenient:

- To specify the default value for a component, you can use the `<>` notation.
- You can use the `|` symbol to give several components the same value.
- You can use the `others` choice to refer to every component that has not yet been specified, provided all those fields have the same type.
- You can use the range notation `..` to refer to specify a contiguous sequence of indices in an array.

However, note that as soon as you used a named association, all subsequent components likewise need to be specified with named associations.

Listing 2: points.ads

```
1 package Points is
2     type Point is record
3         X, Y : Integer := 0;
```

(continues on next page)

(continued from previous page)

```

4  end record;
5
6  type Point_Array is
7    array (Positive range <>) of Point;
8
9  -- use the default values
10 Origin  : Point := (X | Y => <>);
11
12 -- likewise, use the defaults
13 Origin_2 : Point := (others => <>);
14
15 Points_1 : Point_Array := ((1, 2), (3, 4));
16 Points_2 : Point_Array := (1      => (1, 2),
17                           2      => (3, 4),
18                           3 .. 20 => <>);
19 end Points;

```

8.2 Overloading and qualified expressions

Ada has a general concept of name overloading, which we saw earlier in the section on *enumeration types* (page 47).

Let's take a simple example: it is possible in Ada to have functions that have the same name, but different types for their parameters.

Listing 3: pkg.ads

```

1 package Pkg is
2   function F (A : Integer) return Integer;
3   function F (A : Character) return Integer;
4 end Pkg;

```

This is a common concept in programming languages, called *overloading*¹¹, or name overloading.

One of the novel aspects of Ada's overloading facility is the ability to resolve overloading based on the return type of a function.

Listing 4: pkg.ads

```

1 package Pkg is
2   type SSID is new Integer;
3
4   function Convert (Self : SSID) return Integer;
5   function Convert (Self : SSID) return String;
6 end Pkg;

```

Listing 5: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Pkg;          use Pkg;
3
4 procedure Main is
5   S : String := Convert (123_145_299);
6   --           ^ Valid, will choose the
7   --           proper Convert

```

(continues on next page)

¹¹ https://en.wikipedia.org/wiki/Function_overloading

(continued from previous page)

```

8 begin
9   Put_Line (S);
10 end Main;

```

Attention: Note that overload resolution based on the type is allowed for both functions and enumeration literals in Ada - which is why you can have multiple enumeration literals with the same name. Semantically, an enumeration literal is treated like a function that has no parameters.

However, sometimes an ambiguity makes it impossible to resolve which declaration of an overloaded name a given occurrence of the name refers to. This is where a qualified expression becomes useful.

Listing 6: pkg.ads

```

1 package Pkg is
2   type SSID is new Integer;
3
4   function Convert (Self : SSID) return Integer;
5   function Convert (Self : SSID) return String;
6   function Convert (Self : Integer) return String;
7 end Pkg;

```

Listing 7: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Pkg;           use Pkg;
3
4 procedure Main is
5   S : String := Convert (123_145_299);
6   --          ^ Invalid, which convert
7   --          should we call?
8
9   S2 : String := Convert (SSID'(123_145_299));
10  --          ^ We specify that the
11  --          type of the expression
12  --          is SSID.
13
14  -- We could also have declared a temporary
15
16   I : SSID := 123_145_299;
17
18   S3 : String := Convert (I);
19 begin
20   Put_Line (S);
21 end Main;

```

Syntactically the target of a qualified expression can be either any expression in parentheses, or an aggregate:

Listing 8: qual_expr.ads

```

1 package Qual_Expr is
2   type Point is record
3     A, B : Integer;
4   end record;
5
6   P : Point := Point'(12, 15);

```

(continues on next page)

(continued from previous page)

```

7   A : Integer := Integer'(12);
8 end Qual_Expr;
9

```

This illustrates that qualified expressions are a convenient (and sometimes necessary) way for the programmer to make the type of an expression explicit, for the compiler of course, but also for other programmers.

Attention: While they look and feel similar, type conversions and qualified expressions are *not* the same.

A qualified expression specifies the exact type that the target expression will be resolved to, whereas a type conversion will try to convert the target and issue a run-time error if the target value cannot be so converted.

Note that you can use a qualified expression to convert from one subtype to another, with an exception raised if a constraint is violated.

```
X : Integer := Natural'(1);
```

8.3 Character types

As noted earlier, each enumeration type is distinct and incompatible with every other enumeration type. However, what we did not mention previously is that character literals are permitted as enumeration literals. This means that in addition to the language's strongly typed character types, user-defined character types are also permitted:

Listing 9: character_example.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Character_Example is
4    type My_Char is ('a', 'b', 'c');
5    -- Our custom character type, an
6    -- enumeration type with 3 valid values.
7
8    C : Character;
9    -- ^ Built-in character type
10   -- (it's an enumeration type)
11
12   M : My_Char;
13 begin
14   C := '?';
15   -- ^ Character literal
16   -- (enumeration literal)
17
18   M := 'a';
19
20   C := 65;
21   -- ^ Invalid: 65 is not a
22   -- Character value
23
24   C := Character'Val (65);
25   -- Assign the character at
26   -- position 65 in the
27   -- enumeration (which is 'A')

```

(continues on next page)

(continued from previous page)

```

28   M := C;
29   --  ^ Invalid: C is of type Character,
30   --      and M is a My_Char
31
32   M := 'd';
33   --  ^ Invalid: 'd' is not a valid
34   --      literal for type My_Char
35
36 end Character_Example;

```

Build output

```

character_example.adb:14:04: warning: useless assignment to "C", value overwritten
  ↵at line 20 [-gnatwm]
character_example.adb:18:04: warning: useless assignment to "M", value overwritten
  ↵at line 29 [-gnatwm]
character_example.adb:20:04: warning: useless assignment to "C", value overwritten
  ↵at line 24 [-gnatwm]
character_example.adb:20:09: error: expected type "Standard.Character"
character_example.adb:20:09: error: found type universal integer
character_example.adb:29:04: warning: useless assignment to "M", value overwritten
  ↵at line 33 [-gnatwm]
character_example.adb:29:09: error: expected type "My_Char" defined at line 4
character_example.adb:29:09: error: found type "Standard.Character"
character_example.adb:33:04: warning: possibly useless assignment to "M", value
  ↵might not be referenced [-gnatwm]
character_example.adb:33:09: error: character not defined for type "My_Char"
  ↵defined at line 4
gprbuild: *** compilation phase failed

```


ACCESS TYPES (POINTERS)

9.1 Overview

Pointers are a potentially dangerous construct, which conflicts with Ada's underlying philosophy.

There are two ways in which Ada helps shield programmers from the dangers of pointers:

1. One approach, which we have already seen, is to provide alternative features so that the programmer does not need to use pointers. Parameter modes, arrays, and varying size types are all constructs that can replace typical pointer usages in C.
2. Second, Ada has made pointers as safe and restricted as possible, but allows "escape hatches" when the programmer explicitly requests them and presumably will be exercising such features with appropriate care.

Here is how you declare a simple pointer type, or access type, in Ada:

Listing 1: dates.ads

```
1 package Dates is
2   type Months is
3     (January, February, March, April,
4      May, June, July, August, September,
5      October, November, December);
6
7   type Date is record
8     Day : Integer range 1 .. 31;
9     Month : Months;
10    Year : Integer;
11  end record;
12 end Dates;
```

Listing 2: access_types.ads

```
1 with Dates; use Dates;
2
3 package Access_Types is
4   -- Declare an access type
5   type Date_Acc is access Date;
6   --           ^ "Designated type"
7   --           ^ Date_Acc values point
8   --           to Date objects
9
10  D : Date_Acc := null;
11  --           ^ Literal for
12  --           "access to nothing"
13  --           ^ Access to date
14 end Access_Types;
```

This illustrates how to:

- Declare an access type whose values point to ("designate") objects from a specific type
- Declare a variable (access value) from this access type
- Give it a value of **null**

In line with Ada's strong typing philosophy, if you declare a second access type whose designated type is Date, the two access types will be incompatible with each other:

Listing 3: access_types.ads

```
1  with Dates; use Dates;
2
3  package Access_Types is
4      -- Declare an access type
5      type Date_Acc    is access Date;
6      type Date_Acc_2 is access Date;
7
8      D : Date_Acc    := null;
9      D2 : Date_Acc_2 := D;
10     --          ^ Invalid! Different types
11 end Access_Types;
```

Build output

```
access_types.ads:9:24: error: expected type "Date_Acc_2" defined at line 6
access_types.ads:9:24: error: found type "Date_Acc" defined at line 5
gprbuild: *** compilation phase failed
```

In other languages

In most other languages, pointer types are structurally, not nominally typed, like they are in Ada, which means that two pointer types will be the same as long as they share the same target type and accessibility rules.

Not so in Ada, which takes some time getting used to. A seemingly simple problem is, if you want to have a canonical access to a type, where should it be declared? A commonly used pattern is that if you need an access type to a specific type you "own", you will declare it along with the type:

```
package Access_Types is
    type Point is record
        X, Y : Natural;
    end record;

    type Point_Access is access Point;
end Access_Types;
```

9.2 Allocation (by type)

Once we have declared an access type, we need a way to give variables of the types a meaningful value! You can allocate a value of an access type with the `new` keyword in Ada.

Listing 4: access_types.ads

```

1  with Dates; use Dates;
2
3  package Access_Types is
4      type Date_Acc is access Date;
5
6      D : Date_Acc := new Date;
7          --           ^ Allocate a new Date record
8  end Access_Types;
```

If the type you want to allocate needs constraints, you can put them in the subtype indication, just as you would do in a variable declaration:

Listing 5: access_types.ads

```

1  with Dates; use Dates;
2
3  package Access_Types is
4      type String_Acc is access String;
5          ^
6          -- Access to unconstrained array type
7      Msg : String_Acc;
8          --           ^ Default value is null
9
10     Buffer : String_Acc :=
11         new String (1 .. 10);
12             --           ^ Constraint required
13  end Access_Types;
```

Build output

```
access_types.ads:1:06: warning: no entities of "Dates" are referenced [-gnatwu]
access_types.ads:1:13: warning: use clause for package "Dates" has no effect [-
↪gnatwu]
```

In some cases, though, allocating just by specifying the type is not ideal, so Ada also allows you to initialize along with the allocation. This is done via the qualified expression syntax:

Listing 6: access_types.ads

```

1  with Dates; use Dates;
2
3  package Access_Types is
4      type Date_Acc is access Date;
5      type String_Acc is access String;
6
7      D : Date_Acc := new Date'(30, November, 2011);
8      Msg : String_Acc := new String'("Hello");
9  end Access_Types;

```

9.3 Dereferencing

The last important piece of Ada's access type facility is how to get from an access value to the object that is pointed to, that is, how to dereference the pointer. Dereferencing a pointer uses the `.all` syntax in Ada, but is often not needed — in many cases, the access value will be implicitly dereferenced for you:

Listing 7: access_types.ads

```

1  with Dates; use Dates;
2
3  package Access_Types is
4      type Date_Acc is access Date;
5
6      D : Date_Acc := new Date'(30, November, 2011);
7
8      Today : Date := D.all;
9          -- ^ Access value dereference
10     J : Integer := D.Day;
11         -- ^ Implicit dereference for
12         -- record and array components
13         -- Equivalent to D.all.day
14
end Access_Types;

```

9.4 Other features

As you might know if you have used pointers in C or C++, we are still missing features that are considered fundamental to the use of pointers, such as:

- Pointer arithmetic (being able to increment or decrement a pointer in order to point to the next or previous object)
- Manual deallocation - what is called `free` or `delete` in C. This is a potentially unsafe operation. To keep within the realm of safe Ada, you need to never deallocate manually.

Those features exist in Ada, but are only available through specific standard library APIs.

Attention: The guideline in Ada is that most of the time you can avoid manual allocation, and you should.

There are many ways to avoid manual allocation, some of which have been covered (such as parameter modes). The language also provides library abstractions to avoid pointers:

1. One is the use of *containers* (page 191). Containers help users avoid pointers, because container memory is automatically managed.
2. A container to note in this context is the *Indefinite holder*¹². This container allows you to store a value of an indefinite type such as String.
3. GNATCOLL has a library for smart pointers, called *RefCount*¹³. Those pointers' memory is automatically managed, so that when an allocated object has no more references to it, the memory is automatically deallocated.

9.5 Mutually recursive types

The linked list is a common idiom in data structures; in Ada this would be most naturally defined through two types, a record type and an access type, that are mutually dependent. To declare mutually dependent types, you can use an incomplete type declaration:

Listing 8: simple_list.ads

```

1 package Simple_List is
2   type Node;
3   -- This is an incomplete type declaration,
4   -- which is completed in the same
5   -- declarative region.
6
7   type Node_Acc is access Node;
8
9   type Node is record
10    Content : Natural;
11    Prev, Next : Node_Acc;
12  end record;
13 end Simple_List;
```

¹² <http://www.ada-auth.org/standards/12rat/html/Rat12-8-5.html>

¹³ <https://github.com/AdaCore/gnatcoll-core/blob/master/src/gnatcoll-refcount.ads>

MORE ABOUT RECORDS

10.1 Dynamically sized record types

We have previously seen *some simple examples of record types* (page 61). Let's now look at some of the more advanced properties of this fundamental language feature.

One point to note is that object size for a record type does not need to be known at compile time. This is illustrated in the example below:

Listing 1: runtime_length.ads

```
1 package Runtime_Length is
2     function Compute_Max_Len return Natural;
3 end Runtime_Length;
```

Listing 2: var_size_record.ads

```
1 with Runtime_Length; use Runtime_Length;
2
3 package Var_Size_Record is
4     Max_Len : constant Natural
5         := Compute_Max_Len;
6         -- ^ Not known at compile time
7
8     type Items_Array is array (Positive range <>)
9         of Integer;
10
11    type Growable_Stack is record
12        Items : Items_Array (1 .. Max_Len);
13        Len   : Natural;
14    end record;
15    -- Growable_Stack is a definite type, but
16    -- size is not known at compile time.
17
18    G : Growable_Stack;
19 end Var_Size_Record;
```

It is completely fine to determine the size of your records at run time, but note that all objects of this type will have the same size.

10.2 Records with discriminant

In the example above, the size of the `Items` field is determined once, at run-time, but every `Growable_Stack` instance will be exactly the same size. But maybe that's not what you want to do. We saw that arrays in general offer this flexibility: for an unconstrained array type, different objects can have different sizes.

You can get analogous functionality for records, too, using a special kind of field that is called a discriminant:

Listing 3: var_size_record_2.ads

```

1 package Var_Size_Record_2 is
2     type Items_Array is array (Positive range <>)
3         of Integer;
4
5     type Growable_Stack (Max_Len : Natural) is
6         record
7             -- ^ Discriminant. Cannot be
8             -- modified once initialized.
9             Items : Items_Array (1 .. Max_Len);
10            Len   : Natural := 0;
11        end record;
12        -- Growable_Stack is an indefinite type
13        -- (like an array)
14 end Var_Size_Record_2;
```

Discriminants, in their simple forms, are constant: You cannot modify them once you have initialized the object. This intuitively makes sense since they determine the size of the object.

Also, they make a type indefinite: Whether or not the discriminant is used to specify the size of an object, a type with a discriminant will be indefinite if the discriminant is not declared with an initialization:

Listing 4: test_discriminants.ads

```

1 package Test_Discriminants is
2     type Point (X, Y : Natural) is record
3         null;
4     end record;
5
6     P : Point;
7     -- ERROR: Point is indefinite, so you
8     -- need to specify the discriminants
9     -- or give a default value
10
11    P2 : Point (1, 2);
12    P3 : Point := (1, 2);
13    -- Those two declarations are equivalent.
14
15 end Test_Discriminants;
```

Build output

```

test_discriminants.ads:6:08: error: unconstrained subtype not allowed (need
    initialization)
test_discriminants.ads:6:08: error: provide initial value or explicit discriminant
    values
test_discriminants.ads:6:08: error: or give default discriminant values for type
    "Point"
gprbuild: *** compilation phase failed
```

This also means that, in the example above, you cannot declare an array of Point values, because the size of a Point is not known.

As mentioned in the example above, we could provide a default value for the discriminants, so that we could legally declare Point values without specifying the discriminants. For the example above, this is how it would look:

Listing 5: test_discriminants.ads

```

1 package Test_Discriminants is
2   type Point (X, Y : Natural := 0) is record
3     null;
4   end record;
5
6   P : Point;
7   -- We can now simply declare a "Point"
8   -- without further ado. In this case,
9   -- we're using the default values (0)
10  -- for X and Y.
11
12  P2 : Point (1, 2);
13  P3 : Point := (1, 2);
14  -- We can still specify discriminants.
15
16 end Test_Discriminants;
```

Also note that, even though the Point type now has default discriminants, we can still specify discriminants, as we're doing in the declarations of P2 and P3.

In most other respects discriminants behave like regular fields: You have to specify their values in aggregates, as seen above, and you can access their values via the dot notation.

Listing 6: main.adb

```

1 with Var_Size_Record_2; use Var_Size_Record_2;
2 with Ada.Text_Io; use Ada.Text_Io;
3
4 procedure Main is
5   procedure Print_Stack (G : Growable_Stack) is
6   begin
7     Put ("<Stack, items: [");
8     for I in G.Items'Range loop
9       exit when I > G.Len;
10      Put (" " & Integer'Image (G.Items (I)));
11    end loop;
12    Put_Line ("]>");
13  end Print_Stack;
14
15  S : Growable_Stack :=
16    (Max_Len => 128,
17     Items  => (1, 2, 3, 4, others => <>),
18     Len    => 4);
19
20  begin
21    Print_Stack (S);
22  end Main;
```

Build output

```
main.adb:15:04: warning: "S" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
<Stack, items: [ 1 2 3 4]>
```

Note: In the examples above, we used a discriminant to determine the size of an array, but it is not limited to that, and could be used, for example, to determine the size of a nested discriminated record.

10.3 Variant records

The examples of discriminants thus far have illustrated the declaration of records of varying size, by having components whose size depends on the discriminant.

However, discriminants can also be used to obtain the functionality of what are sometimes called "variant records": records that can contain different sets of fields.

Listing 7: variant_record.ads

```

1  package Variant_Record is
2      -- Forward declaration of Expr
3      type Expr;
4
5      -- Access to a Expr
6      type Expr_Access is access Expr;
7
8      type Expr_Kind_Type is (Bin_Op_Plus,
9                               Bin_Op_Minus,
10                              Num);
11     -- A regular enumeration type
12
13     type Expr (Kind : Expr_Kind_Type) is record
14         -- ^ The discriminant is an
15         -- enumeration value
16         case Kind is
17             when Bin_Op_Plus | Bin_Op_Minus =>
18                 Left, Right : Expr_Access;
19             when Num =>
20                 Val : Integer;
21         end case;
22         -- Variant part. Only one, at the end of
23         -- the record definition, but can be
24         -- nested
25     end record;
26 end Variant_Record;
```

The fields that are in a **when** branch will be only available when the value of the discriminant is covered by the branch. In the example above, you will only be able to access the fields **Left** and **Right** when the **Kind** is **Bin_Op_Plus** or **Bin_Op_Minus**.

If you try to access a field that is not valid for your record, a **Constraint_Error** will be raised.

Listing 8: main.adb

```

1  with Variant_Record; use Variant_Record;
2
3  procedure Main is
4      E : Expr := (Num, 12);
5  begin
```

(continues on next page)

(continued from previous page)

```

6   E.Left := new Expr'(Num, 15);
7   -- Will compile but fail at runtime
8 end Main;

```

Build output

```

main.adb:4:04: warning: variable "E" is not referenced [-gnatwu]
main.adb:6:05: warning: component not present in subtype of "Expr" defined at line_4 [enabled by default]
main.adb:6:05: warning: Constraint_Error will be raised at run time [enabled by default]

```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:6 discriminant check failed
```

Here is how you could write an evaluator for expressions:

Listing 9: main.adb

```

1  with Variant_Record; use Variant_Record;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Main is
5    function Eval_Expr (E : Expr) return Integer is
6      (case E.Kind is
7       when Bin_Op_Plus => Eval_Expr (E.Left.all)
8                           + Eval_Expr (E.Right.all),
9       when Bin_Op_Minus => Eval_Expr (E.Left.all)
10                          - Eval_Expr (E.Right.all),
11       when Num => E.Val);
12
13  E : Expr := (Bin_Op_Plus,
14               new Expr'(Bin_Op_Minus,
15                         new Expr'(Num, 12),
16                         new Expr'(Num, 15)),
17               new Expr'(Num, 3));
18 begin
19   Put_Line (Integer'Image (Eval_Expr (E)));
20 end Main;

```

Build output

```
main.adb:13:04: warning: "E" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
0
```

In other languages

Ada's variant records are very similar to Sum types in functional languages such as OCaml or Haskell. A major difference is that the discriminant is a separate field in Ada, whereas the 'tag' of a Sum type is kind of built in, and only accessible with pattern matching.

There are other differences (you can have several discriminants in a variant record in Ada). Nevertheless, they allow the same kind of type modeling as sum types in functional languages.

Compared to C/C++ unions, Ada variant records are more powerful in what they allow, and are also checked at run time, which makes them safer.

FIXED-POINT TYPES

11.1 Decimal fixed-point types

We have already seen how to specify floating-point types. However, in some applications floating-point is not appropriate since, for example, the roundoff error from binary arithmetic may be unacceptable or perhaps the hardware does not support floating-point instructions. Ada provides a category of types, the decimal fixed-point types, that allows the programmer to specify the required decimal precision (number of digits) as well as the scaling factor (a power of ten) and, optionally, a range. In effect the values will be represented as integers implicitly scaled by the specified power of 10. This is useful, for example, for financial applications.

The syntax for a simple decimal fixed-point type is

```
type <type-name> is delta <delta-value> digits <digits-value>;
```

In this case, the **delta** and the **digits** will be used by the compiler to derive a range.

Several attributes are useful for dealing with decimal types:

Attribute Name	Meaning
First	The first value of the type
Last	The last value of the type
Delta	The delta value of the type

In the example below, we declare two data types: T3_D3 and T6_D3. For both types, the delta value is the same: 0.001.

Listing 1: decimal_fixed_point_types.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Decimal_Fixed_Point_Types is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5   type T6_D3 is delta 10.0 ** (-3) digits 6;
6 begin
7   Put_Line ("The delta    value of T3_D3 is "
8             & T3_D3'Image (T3_D3'Delta));
9   Put_Line ("The minimum  value of T3_D3 is "
10            & T3_D3'Image (T3_D3'First));
11   Put_Line ("The maximum  value of T3_D3 is "
12            & T3_D3'Image (T3_D3'Last));
13   New_Line;
14
15   Put_Line ("The delta    value of T6_D3 is "
16             & T6_D3'Image (T6_D3'Delta));
17   Put_Line ("The minimum  value of T6_D3 is "
```

(continues on next page)

(continued from previous page)

```

18      & T6_D3'Image (T6_D3'First));
19  Put_Line ("The maximum value of T6_D3 is "
20      & T6_D3'Image (T6_D3'Last));
21 end Decimal_Fixed_Point_Types;

```

Runtime output

```

The delta value of T3_D3 is 0.001
The minimum value of T3_D3 is -0.999
The maximum value of T3_D3 is 0.999

The delta value of T6_D3 is 0.001
The minimum value of T6_D3 is -999.999
The maximum value of T6_D3 is 999.999

```

When running the application, we see that the delta value of both types is indeed the same: 0.001. However, because T3_D3 is restricted to 3 digits, its range is -0.999 to 0.999. For the T6_D3, we have defined a precision of 6 digits, so the range is -999.999 to 999.999.

Similar to the type definition using the `range` syntax, because we have an implicit range, the compiled code will check that the variables contain values that are not out-of-range. Also, if the result of a multiplication or division on decimal fixed-point types is smaller than the delta value required for the context, the actual result will be zero. For example:

Listing 2: decimal_fixed_point_smaller.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Decimal_Fixed_Point_Smaller is
4   type T3_D3 is delta 10.0 ** (-3) digits 3;
5   type T6_D6 is delta 10.0 ** (-6) digits 6;
6   A : T3_D3 := T3_D3'Delta;
7   B : T3_D3 := 0.5;
8   C : T6_D6;
9 begin
10   Put_Line ("The value of A      is "
11             & T3_D3'Image (A));
12
13   A := A * B;
14   Put_Line ("The value of A * B is "
15             & T3_D3'Image (A));
16
17   A := T3_D3'Delta;
18   C := A * B;
19   Put_Line ("The value of A * B is "
20             & T6_D6'Image (C));
21 end Decimal_Fixed_Point_Smaller;

```

Build output

```

decimal_fixed_point_smaller.adb:7:04: warning: "B" is not modified, could be
→ declared constant [-gnatwk]

```

Runtime output

```

The value of A      is 0.001
The value of A * B is 0.000
The value of A * B is 0.000500

```

In this example, the result of the operation $0.001 * 0.5$ is 0.000500. Since this value is not representable for the T3_D3 type because the delta value is 0.001, the actual value stored

in variable A is zero. However, accuracy is preserved during the arithmetic operations if the target has sufficient precision, and the value displayed for C is 0.000500.

11.2 Ordinary fixed-point types

Ordinary fixed-point types are similar to decimal fixed-point types in that the values are, in effect, scaled integers. The difference between them is in the scale factor: for a decimal fixed-point type, the scaling, given explicitly by the type's **delta**, is always a power of ten.

In contrast, for an ordinary fixed-point type, the scaling is defined by the type's **small**, which is derived from the specified **delta** and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types.

Note: Ordinary fixed-point types can be thought of being closer to the actual representation on the machine, since hardware support for decimal fixed-point arithmetic is not widespread (rescalings by a power of ten), while ordinary fixed-point types make use of the available integer shift instructions.

The syntax for an ordinary fixed-point type is

```
type <type-name> is
  delta <delta-value>
  range <lower-bound> .. <upper-bound>;
```

By default the compiler will choose a scale factor, or **small**, that is a power of 2 no greater than <delta-value>.

For example, we may define a normalized range between -1.0 and 1.0 as following:

Listing 3: normalized_fixed_point_type.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Normalized_Fixed_Point_Type is
4    D : constant := 2.0 ** (-31);
5    type TQ31 is delta D range -1.0 .. 1.0 - D;
6  begin
7    Put_Line ("TQ31 requires "
8              & Integer'Image (TQ31'Size)
9              & " bits");
10   Put_Line ("The delta value of TQ31 is "
11              & TQ31'Image (TQ31'Delta));
12   Put_Line ("The minimum value of TQ31 is "
13              & TQ31'Image (TQ31'First));
14   Put_Line ("The maximum value of TQ31 is "
15              & TQ31'Image (TQ31>Last));
16 end Normalized_Fixed_Point_Type;
```

Runtime output

```
TQ31 requires 32 bits
The delta value of TQ31 is 0.0000000005
The minimum value of TQ31 is -1.0000000000
The maximum value of TQ31 is 0.9999999995
```

In this example, we are defining a 32-bit fixed-point data type for our normalized range. When running the application, we notice that the upper bound is close to one, but not

exact one. This is a typical effect of fixed-point data types — you can find more details in this discussion about the [Q format](#)¹⁴.

We may also rewrite this code with an exact type definition:

Listing 4: normalized_adapted_fixed_point_type.adb

```
1 procedure Normalized_Adapted_Fixed_Point_Type is
2     type TQ31 is
3         delta 2.0 ** (-31)
4         range -1.0 .. 1.0 - 2.0 ** (-31);
5 begin
6     null;
7 end Normalized_Adapted_Fixed_Point_Type;
```

Build output

```
normalized_adapted_fixed_point_type.adb:2:09: warning: type "TQ31" is not u
↳ referenced [-gnatwu]
```

We may also use any other range. For example:

Listing 5: custom_fixed_point_range.adb

```
1 with Ada.Text_Io;  use Ada.Text_Io;
2 with Ada.Numerics; use Ada.Numerics;
3
4 procedure Custom_Fixed_Point_Range is
5     type T_Inv_Trig is
6         delta 2.0 ** (-15) * Pi
7         range -Pi / 2.0 .. Pi / 2.0;
8 begin
9     Put_Line ("T_Inv_Trig requires "
10             & Integer'Image (T_Inv_Trig'Size)
11             & " bits");
12     Put_Line ("The delta value of T_Inv_Trig is "
13             & T_Inv_Trig'Image (T_Inv_Trig'Delta));
14     Put_Line ("The minimum value of T_Inv_Trig is "
15             & T_Inv_Trig'Image (T_Inv_Trig'First));
16     Put_Line ("The maximum value of T_Inv_Trig is "
17             & T_Inv_Trig'Image (T_Inv_Trig'Last));
18 end Custom_Fixed_Point_Range;
```

Build output

```
custom_fixed_point_range.adb:13:44: warning: static fixed-point value is not a u
↳ multiple of Small [-gnatwb]
```

Runtime output

```
T_Inv_Trig requires 16 bits
The delta value of T_Inv_Trig is 0.00006
The minimum value of T_Inv_Trig is -1.57080
The maximum value of T_Inv_Trig is 1.57080
```

In this example, we are defining a 16-bit type called `T_Inv_Trig`, which has a range from $-\pi/2$ to $\pi/2$.

All standard operations are available for fixed-point types. For example:

¹⁴ [https://en.wikipedia.org/wiki/Q_\(number_format\)](https://en.wikipedia.org/wiki/Q_(number_format))

Listing 6: fixed_point_op.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Fixed_Point_Op is
4      type TQ31 is
5          delta 2.0 ** (-31)
6          range -1.0 .. 1.0 - 2.0 ** (-31);
7
8      A, B, R : TQ31;
9  begin
10     A := 0.25;
11     B := 0.50;
12     R := A + B;
13     Put_Line ("R is " & TQ31'Image (R));
14 end Fixed_Point_Op;

```

Runtime output

```
R is 0.7500000000
```

As expected, R contains 0.75 after the addition of A and B.

In fact the language is more general than these examples imply, since in practice it is typical to need to multiply or divide values from different fixed-point types, and obtain a result that may be of a third fixed-point type. The details are outside the scope of this introductory course.

It is also worth noting, although again the details are outside the scope of this course, that you can explicitly specify a value for an ordinary fixed-point type's small. This allows non-binary scaling, for example:

```

type Angle is
    delta 1.0/3600.0
    range 0.0 .. 360.0 - 1.0 / 3600.0;
for Angle'Small use Angle'Delta;

```


PRIVACY

One of the main principles of modular programming, as well as object oriented programming, is [encapsulation¹⁵](#).

Encapsulation, briefly, is the concept that the implementer of a piece of software will distinguish between the code's public interface and its private implementation.

This is not only applicable to software libraries but wherever abstraction is used.

In Ada, the granularity of encapsulation is a bit different from most object-oriented languages, because privacy is generally specified at the package level.

12.1 Basic encapsulation

Listing 1: encapsulate.ads

```
1 package Encapsulate is
2     procedure Hello;
3
4     private
5
6         procedure Hello2;
7         -- Not visible from external units
8     end Encapsulate;
```

Listing 2: encapsulate.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Encapsulate is
4
5     procedure Hello is
6     begin
7         Put_Line ("Hello");
8     end Hello;
9
10    procedure Hello2 is
11    begin
12        Put_Line ("Hello #2");
13    end Hello2;
14
15 end Encapsulate;
```

¹⁵ [https://en.wikipedia.org/wiki/Encapsulation_\(computer_programming\)](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))

Listing 3: main.adb

```

1  with Encapsulate;
2
3  procedure Main is
4  begin
5      Encapsulate.Hello;
6      Encapsulate.Hello2;
7      -- Invalid: Hello2 is not visible
8  end Main;

```

Build output

```

main.adb:6:15: error: "Hello2" is not a visible entity of "Encapsulate"
gprbuild: *** compilation phase failed

```

12.2 Abstract data types

With this high-level granularity, it might not seem obvious how to hide the implementation details of a type. Here is how it can be done in Ada:

Listing 4: stacks.ads

```

1  package Stacks is
2      type Stack is private;
3          -- Declare a private type: You cannot depend
4          -- on its implementation. You can only assign
5          -- and test for equality.
6
7      procedure Push (S    : in out Stack;
8                      Val   :           Integer);
9      procedure Pop  (S    : in out Stack;
10                      Val   :           out Integer);
11  private
12
13      subtype Stack_Index is Natural range 1 .. 10;
14      type Content_Type is array (Stack_Index)
15          of Natural;
16
17      type Stack is record
18          Top    : Stack_Index;
19          Content : Content_Type;
20      end record;
21  end Stacks;

```

Listing 5: stacks.adb

```

1  package body Stacks is
2
3      procedure Push (S    : in out Stack;
4                      Val   :           Integer) is
5      begin
6          -- Missing implementation!
7          null;
8      end Push;
9
10     procedure Pop  (S    : in out Stack;
11                      Val   :           out Integer) is

```

(continues on next page)

(continued from previous page)

```

12  begin
13      -- Dummy implementation!
14      Val := 0;
15  end Pop;
16
17 end Stacks;
```

In the above example, we define a stack type in the public part (known as the *visible part* of the package spec in Ada), but the exact representation of that type is private.

Then, in the private part, we define the representation of that type. We can also declare other types that will be used as *helpers* for our main public type. This is useful since declaring helper types is common in Ada.

A few words about terminology:

- The Stack type as viewed from the public part is called the partial view of the type. This is what clients have access to.
- The Stack type as viewed from the private part or the body of the package is called the full view of the type. This is what implementers have access to.

From the point of view of the client (the *with'ing* unit), only the public (visible) part is important, and the private part could as well not exist. It makes it very easy to read linearly the part of the package that is important for you.

```

-- No need to read the private part to use the package
package Stacks is
    type Stack is private;

    procedure Push (S : in out Stack;
                   Val :           Integer);
    procedure Pop  (S : in out Stack;
                   Val :       out Integer);
private
    ...
end Stacks;
```

Here is how the Stacks package would be used:

```

-- Example of use
with Stacks; use Stacks;

procedure Test_Stack is
    S : Stack;
    Res : Integer;
begin
    Push (S, 5);
    Push (S, 7);
    Pop  (S, Res);
end Test_Stack;
```

12.3 Limited types

Ada's *limited type* facility allows you to declare a type for which assignment and comparison operations are not automatically provided.

Listing 6: stacks.ads

```

1 package Stacks is
2   type Stack is limited private;
3   -- Limited type. Cannot assign nor compare.
4
5   procedure Push (S    : in out Stack;
6                   Val :          Integer);
7   procedure Pop  (S    : in out Stack;
8                   Val :          out Integer);
9
10  private
11    subtype Stack_Index is Natural range 1 .. 10;
12    type Content_Type is
13      array (Stack_Index) of Natural;
14
15    type Stack is limited record
16      Top    : Stack_Index;
17      Content : Content_Type;
18    end record;
19 end Stacks;

```

Listing 7: stacks.adb

```

1 package body Stacks is
2
3   procedure Push (S    : in out Stack;
4                   Val :          Integer) is
5   begin
6     -- Missing implementation!
7     null;
8   end Push;
9
10  procedure Pop  (S    : in out Stack;
11                   Val :          out Integer) is
12  begin
13    -- Dummy implementation!
14    Val := 0;
15  end Pop;
16
17 end Stacks;

```

Listing 8: main.adb

```

1 with Stacks; use Stacks;
2
3 procedure Main is
4   S, S2 : Stack;
5 begin
6   S := S2;
7   -- Illegal: S is limited.
8 end Main;

```

Build output

```
stacks.adb:10:19: warning: formal parameter "S" is not referenced [-gnatwf]
(continues on next page)
```

(continued from previous page)

```
main.adb:6:04: error: left hand of assignment must not be limited type
gprbuild: *** compilation phase failed
```

This is useful because, for example, for some data types the built-in assignment operation might be incorrect (for example when a deep copy is required).

Ada does allow you to overload the comparison operators = and /= for limited types (and to override the built-in declarations for non-limited types).

Ada also allows you to implement special semantics for assignment via controlled types¹⁶. However, in some cases assignment is simply inappropriate; one example is the **File_Type** from the Ada.Text_Io package, which is declared as a limited type and thus attempts to assign one file to another would be detected as illegal.

12.4 Child packages & privacy

We've seen previously (in the *child packages section* (page 35)) that packages can have child packages. Privacy plays an important role in child packages. This section discusses some of the privacy rules that apply to child packages.

Although the private part of a package P is meant to encapsulate information, certain parts of a child package P.C can have access to this private part of P. In those cases, information from the private part of P can then be used as if it were declared in the public part of its specification. To be more specific, the body of P.C and the private part of the specification of P.C have access to the private part of P. However, the public part of the specification of P.C only has access to the public part of P's specification. The following table summarizes this:

Part of a child package	Access to the private part of its parent's specification
Specification: public part	
Specification: private part	✓
Body	✓

The rest of this section shows examples of how this access to private information actually works for child packages.

Let's first look at an example where the body of a child package P.C has access to the private part of the specification of its parent P. We've seen, in a previous source-code example, that the Hello2 procedure declared in the private part of the Encapsulate package cannot be used in the Main procedure, since it's not visible there. This limitation doesn't apply, however, for parts of the child packages of the Encapsulate package. In fact, the body of its child package Encapsulate.Child has access to the Hello2 procedure and can call it there, as you can see in the implementation of the Hello3 procedure of the Child package:

Listing 9: encapsulate.ads

```

1 package Encapsulate is
2   procedure Hello;
3
4   private
5
6     procedure Hello2;
7     -- Not visible from external units
8     -- But visible in child packages
9   end Encapsulate;
```

¹⁶ <http://www.adা-auth.org/standards/12rm/html/RM-7-6.html>

Listing 10: encapsulate.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Encapsulate is
4
5   procedure Hello is
6   begin
7     Put_Line ("Hello");
8   end Hello;
9
10  procedure Hello2 is
11  begin
12    Put_Line ("Hello #2");
13  end Hello2;
14
15 end Encapsulate;
```

Listing 11: encapsulate-child.ads

```
1 package Encapsulate.Child is
2
3   procedure Hello3;
4
5 end Encapsulate.Child;
```

Listing 12: encapsulate-child.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Encapsulate.Child is
4
5   procedure Hello3 is
6   begin
7     -- Using private procedure Hello2
8     -- from the parent package
9     Hello2;
10    Put_Line ("Hello #3");
11  end Hello3;
12
13 end Encapsulate.Child;
```

Listing 13: main.adb

```
1 with Encapsulate.Child;
2
3 procedure Main is
4 begin
5   Encapsulate.Child.Hello3;
6 end Main;
```

Runtime output

```
Hello #2
Hello #3
```

The same mechanism applies to types declared in the private part of a parent package. For instance, the body of a child package can access components of a record declared in the private part of its parent package. Let's look at an example:

Listing 14: my_types.ads

```

1 package My_Types is
2
3     type Priv_Rec is private;
4
5 private
6
7     type Priv_Rec is record
8         Number : Integer := 42;
9     end record;
10
11 end My_Types;

```

Listing 15: my_types-ops.ads

```

1 package My_Types.Ops is
2
3     procedure Display (E : Priv_Rec);
4
5 end My_Types.Ops;

```

Listing 16: my_types-ops.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Types.Ops is
4
5     procedure Display (E : Priv_Rec) is
6     begin
7         Put_Line ("Priv_Rec.Number: "
8                   & Integer'Image (E.Number));
9     end Display;
10
11 end My_Types.Ops;

```

Listing 17: main.adb

```

1 with Ada.Text_IO;  use Ada.Text_IO;
2
3 with My_Types;    use My_Types;
4 with My_Types.Ops; use My_Types.Ops;
5
6 procedure Main is
7     E : Priv_Rec;
8 begin
9     Put_Line ("Presenting information:");
10
11     -- The following code would trigger a
12     -- compilation error here:
13     --
14     -- Put_Line ("Priv_Rec.Number: "
15     --           & Integer'Image (E.Number));
16
17     Display (E);
18 end Main;

```

Runtime output

```

Presenting information:
Priv_Rec.Number: 42

```

In this example, we don't have access to the Number component of the record type `Priv_Rec` in the `Main` procedure. You can see this in the call to `Put_Line` that has been commented-out in the implementation of `Main`. Trying to access the Number component there would trigger a compilation error. But we do have access to this component in the body of the `My_Types.Ops` package, since it's a child package of the `My_Types` package. Therefore, `Ops`'s body has access to the declaration of the `Priv_Rec` type — which is in the private part of its parent, the `My_Types` package. For this reason, the same call to `Put_Line` that would trigger a compilation error in the `Main` procedure works fine in the `Display` procedure of the `My_Types.Ops` package.

This kind of privacy rules for child packages allows for extending the functionality of a parent package and, at the same time, retain its encapsulation.

As we mentioned previously, in addition to the package body, the private part of the specification of a child package `P.C` also has access to the private part of the specification of its parent `P`. Let's look at an example where we declare an object of private type `Priv_Rec` in the private part of the child package `My_Types.Child` and initialize the Number component of the `Priv_Rec` record directly:

```
package My_Types.Child is
  private
    E : Priv_Rec := (Number => 99);
end My_Types.Ops;
```

As expected, we wouldn't be able to initialize this component if we moved this declaration to the public (visible) part of the same child package:

```
package My_Types.Child is
  E : Priv_Rec := (Number => 99);
end My_Types.Ops;
```

The declaration above triggers a compilation error, since type `Priv_Rec` is private. Because the public part of `My_Types.Child` is also visible outside the child package, Ada cannot allow accessing private information in this part of the specification.

GENERICs

13.1 Introduction

Generics are used for metaprogramming in Ada. They are useful for abstract algorithms that share common properties with each other.

Either a subprogram or a package can be generic. A generic is declared by using the keyword **generic**. For example:

Listing 1: operator.ads

```
1 generic
2   type T is private;
3   -- Declaration of formal types and objects
4   -- Below, we could use one of the following:
5   -- <procedure | function | package>
6   procedure Operator (Dummy : in out T);
```

Listing 2: operator.adb

```
1 procedure Operator (Dummy : in out T) is
2 begin
3   null;
4 end Operator;
```

13.2 Formal type declaration

Formal types are abstractions of a specific type. For example, we may want to create an algorithm that works on any integer type, or even on any type at all, whether a numeric type or not. The following example declares a formal type T for the Set procedure.

Listing 3: set.ads

```
1 generic
2   type T is private;
3   -- T is a formal type that indicates that
4   -- any type can be used, possibly a numeric
5   -- type or possibly even a record type.
6   procedure Set (Dummy : T);
```

Listing 4: set.adb

```
1 procedure Set (Dummy : T) is
2 begin
```

(continues on next page)

(continued from previous page)

```

3   null;
4 end Set;

```

The declaration of T as **private** indicates that you can map any definite type to it. But you can also restrict the declaration to allow only some types to be mapped to that formal type. Here are some examples:

Formal Type	Format
Any type	type T is private;
Any discrete type	type T is (<>);
Any floating-point type	type T is digits <>;

13.3 Formal object declaration

Formal objects are similar to subprogram parameters. They can reference formal types declared in the formal specification. For example:

Listing 5: set.ads

```

1 generic
2   type T is private;
3   X : in out T;
4   -- X can be used in the Set procedure
5 procedure Set (E : T);

```

Listing 6: set.adb

```

1 procedure Set (E : T) is
2   pragma Unreferenced (E, X);
3 begin
4   null;
5 end Set;

```

Formal objects can be either input parameters or specified using the **in out** mode.

13.4 Generic body definition

We don't repeat the **generic** keyword for the body declaration of a generic subprogram or package. Instead, we start with the actual declaration and use the generic types and objects we declared. For example:

Listing 7: set.ads

```

1 generic
2   type T is private;
3   X : in out T;
4 procedure Set (E : T);

```

Listing 8: set.adb

```

1 procedure Set (E : T) is
2   -- Body definition: "generic" keyword
3   -- is not used

```

(continues on next page)

(continued from previous page)

```

4 begin
5   X := E;
6 end Set;
```

13.5 Generic instantiation

Generic subprograms or packages can't be used directly. Instead, they need to be instantiated, which we do using the `new` keyword, as shown in the following example:

Listing 9: set.ads

```

1 generic
2   type T is private;
3   X : in out T;
4   -- X can be used in the Set procedure
5 procedure Set (E : T);
```

Listing 10: set.adb

```

1 procedure Set (E : T) is
2 begin
3   X := E;
4 end Set;
```

Listing 11: show_generic_instantiation.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Set;
3
4 procedure Show_Generic_Instantiation is
5
6   Main      : Integer := 0;
7   Current   : Integer;
8
9   procedure Set_Main is new Set (T => Integer,
10                                X => Main);
11   -- Here, we map the formal parameters to
12   -- actual types and objects.
13   --
14   -- The same approach can be used to
15   -- instantiate functions or packages, e.g.:
16   --
17   -- function Get_Main is new ...
18   -- package Integer_Queue is new ...
19
20 begin
21   Current := 10;
22
23   Set_Main (Current);
24   Put_Line ("Value of Main is "
25             & Integer'Image (Main));
26 end Show_Generic_Instantiation;
```

Runtime output

```
Value of Main is 10
```

In the example above, we instantiate the procedure Set by mapping the formal parameters T and X to actual existing elements, in this case the **Integer** type and the Main variable.

13.6 Generic packages

The previous examples focused on generic subprograms. In this section, we look at generic packages. The syntax is similar to that used for generic subprograms: we start with the **generic** keyword and continue with formal declarations. The only difference is that **package** is specified instead of a subprogram keyword.

Here's an example:

Listing 12: element.ads

```
1 generic
2   type T is private;
3 package Element is
4
5   procedure Set (E : T);
6   procedure Reset;
7   function Get return T;
8   function Is_Valid return Boolean;
9
10  Invalid_Element : exception;
11
12 private
13   Value : T;
14   Valid : Boolean := False;
15 end Element;
```

Listing 13: element.adb

```
1 package body Element is
2
3   procedure Set (E : T) is
4   begin
5     Value := E;
6     Valid := True;
7   end Set;
8
9   procedure Reset is
10  begin
11    Valid := False;
12  end Reset;
13
14   function Get return T is
15   begin
16     if not Valid then
17       raise Invalid_Element;
18     end if;
19     return Value;
20   end Get;
21
22   function Is_Valid return Boolean is (Valid);
23 end Element;
```

Listing 14: show_generic_package.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2  with Element;
3
4  procedure Show_Generic_Package is
5
6    package I is new Element (T => Integer);
7
8    procedure Display_Initialized is
9    begin
10      if I.Is_Valid then
11        Put_Line ("Value is initialized");
12      else
13        Put_Line ("Value is not initialized");
14      end if;
15    end Display_Initialized;
16
17  begin
18    Display_Initialized;
19
20    Put_Line ("Initializing...");
21    I.Set (5);
22    Display_Initialized;
23    Put_Line ("Value is now set to "
24              & Integer'Image (I.Get));
25
26    Put_Line ("Resetting...");
27    I.Reset;
28    Display_Initialized;
29
30  end Show_Generic_Package;

```

Runtime output

```

Value is not initialized
Initializing...
Value is initialized
Value is now set to 5
Resetting...
Value is not initialized

```

In the example above, we created a simple container named `Element`, with just one single element. This container tracks whether the element has been initialized or not.

After writing package definition, we create the instance `I` of the `Element`. We use the instance by calling the package subprograms (`Set`, `Reset`, and `Get`).

13.7 Formal subprograms

In addition to formal types and objects, we can also declare formal subprograms or packages. This course only describes formal subprograms; formal packages are discussed in the advanced course.

We use the `with` keyword to declare a formal subprogram. In the example below, we declare a formal function (`Comparison`) to be used by the generic procedure `Check`.

Listing 15: check.ads

```
1 generic
2     Description : String;
3     type T is private;
4     with function Comparison (X, Y : T) return Boolean;
5 procedure Check (X, Y : T);
```

Listing 16: check.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Check (X, Y : T) is
4     Result : Boolean;
5 begin
6     Result := Comparison (X, Y);
7     if Result then
8         Put_Line ("Comparison (" & Description & ") between arguments is OK!");
9     else
10        Put_Line ("Comparison (" & Description & ") between arguments is not OK!");
11    end if;
12 end Check;
```

Listing 17: show_formal_subprogram.adb

```
1 with Check;
2
3 procedure Show_Formal_Subprogram is
4
5     A, B : Integer;
6
7     procedure Check_Is_Equal is new
8         Check (Description => "equality",
9                 T          => Integer,
10                Comparison => Standard."=");
11    -- Here, we are mapping the standard
12    -- equality operator for Integer types to
13    -- the Comparison formal function
14 begin
15     A := 0;
16     B := 1;
17     Check_Is_Equal (A, B);
18 end Show_Formal_Subprogram;
```

Runtime output

```
Comparison (equality) between arguments is not OK!
```

13.8 Example: I/O instances

Ada offers generic I/O packages that can be instantiated for standard and derived types. One example is the generic `Float_Io` package, which provides procedures such as `Put` and `Get`. In fact, `Float_Text_Io` — available from the standard library — is an instance of the `Float_Io` package, and it's defined as:

```
with Ada.Text_Io;

package Ada.Float_Text_Io is new Ada.Text_Io.Float_Io (Float);
```

You can use it directly with any object of floating-point type. For example:

Listing 18: `show_float_text_io.adb`

```
1  with Ada.Float_Text_Io;
2
3  procedure Show_Float_Text_Io is
4      X : constant Float := 2.5;
5
6      use Ada.Float_Text_Io;
7  begin
8      Put (X);
9  end Show_Float_Text_Io;
```

Runtime output

```
2.50000E+00
```

Instantiating generic I/O packages can be useful for derived types. For example, let's create a new type `Price` that must be displayed with two decimal digits after the point, and no exponent.

Listing 19: `show_float_io_inst.adb`

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Float_Io_Inst is
4
5      type Price is digits 3;
6
7      package Price_Io is new
8          Ada.Text_Io.Float_Io (Price);
9
10     P : Price;
11  begin
12      -- Set to zero => don't display exponent
13      Price_Io.Default_Exp := 0;
14
15      P := 2.5;
16      Price_Io.Put (P);
17      New_Line;
18
19      P := 5.75;
20      Price_Io.Put (P);
21      New_Line;
22  end Show_Float_Io_Inst;
```

Runtime output

```
2.50
5.75
```

By adjusting Default_Exp from the Price_I0 instance to *remove* the exponent, we can control how variables of Price type are displayed. Just as a side note, we could also have written:

```
-- [ ... ]

type Price is new Float;

package Price_I0 is new
  Ada.Text_Io.Float_Io (Price);

begin
  Price_I0.Default_Aft := 2;
  Price_I0.Default_Exp := 0;
```

In this case, we're adjusting Default_Aft, too, to get two decimal digits after the point when calling Put.

In addition to the generic Float_Io package, the following generic packages are available from Ada.Text_Io:

- Enumeration_Io for enumeration types;
- Integer_Io for integer types;
- Modular_Io for modular types;
- Fixed_Io for fixed-point types;
- Decimal_Io for decimal types.

In fact, we could rewrite the example above using decimal types:

Listing 20: show_decimal_io_inst.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Decimal_Io_Inst is
4
5    type Price is delta 10.0 ** (-2) digits 12;
6
7    package Price_I0 is new
8      Ada.Text_Io.Decimal_Io (Price);
9
10   P : Price;
11 begin
12   Price_I0.Default_Exp := 0;
13
14   P := 2.5;
15   Price_I0.Put (P);
16   New_Line;
17
18   P := 5.75;
19   Price_I0.Put (P);
20   New_Line;
21 end Show_Decimal_Io_Inst;
```

Runtime output

```
2.50
5.75
```

13.9 Example: ADTs

An important application of generics is to model abstract data types (ADTs). In fact, Ada includes a library with numerous ADTs using generics: Ada.Containers (described in the [containers section](#) (page 191)).

A typical example of an ADT is a stack:

Listing 21: stacks.ads

```

1 generic
2   Max : Positive;
3   type T is private;
4 package Stacks is
5
6   type Stack is limited private;
7
8   Stack_Underflow, Stack_Overflow : exception;
9
10  function Is_Empty (S : Stack) return Boolean;
11
12  function Pop (S : in out Stack) return T;
13
14  procedure Push (S : in out Stack;
15                V :          T);
16
17 private
18
19  type Stack_Array is
20    array (Natural range <>) of T;
21
22  Min : constant := 1;
23
24  type Stack is record
25    Container : Stack_Array (Min .. Max);
26    Top       : Natural := Min - 1;
27  end record;
28
29 end Stacks;

```

Listing 22: stacks.adb

```

1 package body Stacks is
2
3   function Is_Empty (S : Stack) return Boolean is
4     (S.Top < S.Container'First);
5
6   function Is_Full (S : Stack) return Boolean is
7     (S.Top >= S.Container'Last);
8
9   function Pop (S : in out Stack) return T is
10 begin
11   if Is_Empty (S) then
12     raise Stack_Underflow;
13   else
14     return X : T do
15       X      := S.Container (S.Top);
16       S.Top := S.Top - 1;
17     end return;
18   end if;
19 end Pop;

```

(continues on next page)

(continued from previous page)

```

20
21 procedure Push (S : in out Stack;
22                  V :          T) is
23 begin
24   if Is_Full (S) then
25     raise Stack_Overflow;
26   else
27     S.Top           := S.Top + 1;
28     S.Container (S.Top) := V;
29   end if;
30 end Push;
31
32 end Stacks;

```

Listing 23: show_stack.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Stacks;
3
4  procedure Show_Stack is
5
6    package Integer_Stacks is new
7      Stacks (Max => 10,
8              T    => Integer);
9    use Integer_Stacks;
10
11   Values : Integer_Stacks.Stack;
12
13 begin
14   Push (Values, 10);
15   Push (Values, 20);
16
17   Put_Line ("Last value was "
18             & Integer'Image (Pop (Values)));
19 end Show_Stack;

```

Runtime output

```
Last value was 20
```

In this example, we first create a generic stack package (Stacks) and then instantiate it to create a stack of up to 10 integer values.

13.10 Example: Swap

Let's look at a simple procedure that swaps variables of type Color:

Listing 24: colors.ads

```

1  package Colors is
2    type Color is (Black, Red, Green,
3                  Blue, White);
4
5    procedure Swap_Colors (X, Y : in out Color);
6  end Colors;

```

Listing 25: colors.adb

```

1 package body Colors is
2
3     procedure Swap_Colors (X, Y : in out Color) is
4         Tmp : constant Color := X;
5     begin
6         X := Y;
7         Y := Tmp;
8     end Swap_Colors;
9
10 end Colors;

```

Listing 26: test_non_generic_swap_colors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Test_Non_Generic_Swap_Colors is
5     A, B, C : Color;
6 begin
7     A := Blue;
8     B := White;
9     C := Red;
10
11    Put_Line ("Value of A is "
12              & Color'Image (A));
13    Put_Line ("Value of B is "
14              & Color'Image (B));
15    Put_Line ("Value of C is "
16              & Color'Image (C));
17
18    New_Line;
19    Put_Line ("Swapping A and C... ");
20    New_Line;
21    Swap_Colors (A, C);
22
23    Put_Line ("Value of A is "
24              & Color'Image (A));
25    Put_Line ("Value of B is "
26              & Color'Image (B));
27    Put_Line ("Value of C is "
28              & Color'Image (C));
29 end Test_Non_Generic_Swap_Colors;

```

Runtime output

```

Value of A is BLUE
Value of B is WHITE
Value of C is RED

Swapping A and C...

Value of A is RED
Value of B is WHITE
Value of C is BLUE

```

In this example, `Swap_Colors` can only be used for the `Color` type. However, this algorithm can theoretically be used for any type, whether an enumeration type or a complex record type with many elements. The algorithm itself is the same: it's only the type that differs. If, for example, we want to swap variables of `Integer` type, we don't want to duplicate the

implementation. Therefore, such an algorithm is a perfect candidate for abstraction using generics.

In the example below, we create a generic version of Swap_Colors and name it **Generic_Swap**. This generic version can operate on any type due to the declaration of formal type T.

Listing 27: generic_swap.ads

```
1 generic
2     type T is private;
3 procedure Generic_Swap (X, Y : in out T);
```

Listing 28: generic_swap.adb

```
1 procedure Generic_Swap (X, Y : in out T) is
2     Tmp : constant T := X;
3 begin
4     X := Y;
5     Y := Tmp;
6 end Generic_Swap;
```

Listing 29: colors.ads

```
1 with Generic_Swap;
2
3 package Colors is
4
5     type Color is (Black, Red, Green,
6                     Blue, White);
7
8     procedure Swap_Colors is new
9         Generic_Swap (T => Color);
10
11 end Colors;
```

Listing 30: test_swap_colors.adb

```
1 with Ada.Text_Io;  use Ada.Text_Io;
2 with Colors;       use Colors;
3
4 procedure Test_Swap_Colors is
5     A, B, C : Color;
6 begin
7     A := Blue;
8     B := White;
9     C := Red;
10
11     Put_Line ("Value of A is "
12               & Color'Image (A));
13     Put_Line ("Value of B is "
14               & Color'Image (B));
15     Put_Line ("Value of C is "
16               & Color'Image (C));
17
18     New_Line;
19     Put_Line ("Swapping A and C....");
20     New_Line;
21     Swap_Colors (A, C);
22
23     Put_Line ("Value of A is "
```

(continues on next page)

(continued from previous page)

```

24      & Color'Image (A));
25  Put_Line ("Value of B is "
26            & Color'Image (B));
27  Put_Line ("Value of C is "
28            & Color'Image (C));
29 end Test_Swap_Colors;

```

Runtime output

```

Value of A is BLUE
Value of B is WHITE
Value of C is RED

Swapping A and C...

Value of A is RED
Value of B is WHITE
Value of C is BLUE

```

As we can see in the example, we can create the same Swap_Colors procedure as we had in the non-generic version of the algorithm by declaring it as an instance of the generic **Generic_Swap** procedure. We specify that the generic T type will be mapped to the Color type by passing it as an argument to the **Generic_Swap** instantiation.

13.11 Example: Reversing

The previous example, with an algorithm to swap two values, is one of the simplest examples of using generics. Next we study an algorithm for reversing elements of an array. First, let's start with a non-generic version of the algorithm, one that works specifically for the Color type:

Listing 31: colors.ads

```

1 package Colors is
2
3   type Color is (Black, Red, Green,
4                  Blue, White);
5
6   type Color_Array is
7     array (Integer range <>) of Color;
8
9   procedure Reverse_It (X : in out Color_Array);
10
11 end Colors;

```

Listing 32: colors.adb

```

1 package body Colors is
2
3   procedure Reverse_It (X : in out Color_Array) is
4 begin
5     for I in X'First ..
6       (X'Last + X'First) / 2 loop
7       declare
8         Tmp      : Color;
9         X_Left  : Color
10        renames X (I);

```

(continues on next page)

(continued from previous page)

```

11      X_Right : Color
12          renames X (X'Last + X'First - I);
13  begin
14      Tmp      := X_Left;
15      X_Left   := X_Right;
16      X_Right := Tmp;
17  end;
18  end loop;
19 end Reverse_It;
20
21 end Colors;

```

Listing 33: test_non_generic_reverse_colors.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2 with Colors;      use Colors;
3
4 procedure Test_Non_Generic_Reverse_Colors is
5
6     My_Colors : Color_Array (1 .. 5) :=
7         (Black, Red, Green, Blue, White);
8
9 begin
10    for C of My_Colors loop
11        Put_Line ("My_Color: " & Color'Image (C));
12    end loop;
13
14    New_Line;
15    Put_Line ("Reversing My_Color...");
16    New_Line;
17    Reverse_It (My_Colors);
18
19    for C of My_Colors loop
20        Put_Line ("My_Color: " & Color'Image (C));
21    end loop;
22
23 end Test_Non_Generic_Reverse_Colors;

```

Runtime output

```

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK

```

The procedure `Reverse_It` takes an array of colors, starts by swapping the first and last elements of the array, and continues doing that with successive elements until it reaches the middle of array. At that point, the entire array has been reversed, as we see from the output of the test program.

To abstract this procedure, we declare formal types for three components of the algorithm:

- the elements of the array (Color type in the example)

- the range used for the array (**Integer** range in the example)
- the actual array type (Color_Array type in the example)

This is a generic version of the algorithm:

Listing 34: generic_reverse.ads

```

1 generic
2   type T is private;
3   type Index is range <>;
4   type Array_T is
5     array (Index range <>) of T;
6 procedure Generic_Reverse (X : in out Array_T);

```

Listing 35: generic_reverse.adb

```

1 procedure Generic_Reverse (X : in out Array_T) is
2 begin
3   for I in X'First .. (X'Last + X'First) / 2 loop
4     declare
5       Tmp      : T;
6       X_Left  : T
7       renames X (I);
8       X_Right : T
9       renames X (X'Last + X'First - I);
10    begin
11      Tmp := X_Left;
12      X_Left := X_Right;
13      X_Right := Tmp;
14    end;
15  end loop;
16 end Generic_Reverse;
17

```

Listing 36: colors.ads

```

1 with Generic_Reverse;
2
3 package Colors is
4
5   type Color is (Black, Red, Green,
6                 Blue, White);
7
8   type Color_Array is
9     array (Integer range <>) of Color;
10
11  procedure Reverse_It is new
12    Generic_Reverse (T      => Color,
13                     Index  => Integer,
14                     Array_T => Color_Array);
15
16 end Colors;
17

```

Listing 37: test_reverse_colors.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Colors;      use Colors;
3
4 procedure Test_Reverse_Colors is
5
6   My_Colors : Color_Array (1 .. 5) :=

```

(continues on next page)

(continued from previous page)

```

7      (Black, Red, Green, Blue, White);
8
9 begin
10   for C of My_Colors loop
11     Put_Line ("My_Color: "
12       & Color'Image (C));
13   end loop;
14
15   New_Line;
16   Put_Line ("Reversing My_Color... ");
17   New_Line;
18   Reverse_It (My_Colors);
19
20   for C of My_Colors loop
21     Put_Line ("My_Color: "
22       & Color'Image (C));
23   end loop;
24
25 end Test_Reverse_Colors;

```

Runtime output

```

My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Reversing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK

```

As mentioned above, we're abstracting three components of the algorithm:

- the T type abstracts the elements of the array
- the Index type abstracts the range used for the array
- the Array_T type abstracts the array type and uses the formal declarations of the T and Index types.

13.12 Example: Test application

In the previous example we've focused only on abstracting the reversing algorithm itself. However, we could have decided to also abstract our small test application. This could be useful if we, for example, decide to test other procedures that change elements of an array.

In order to do this, we again have to choose the elements to abstract. We therefore declare the following formal parameters:

- S: the string containing the array name
- a function Image that converts an element of type T to a string
- a procedure Test that performs some operation on the array

Note that Image and Test are examples of formal subprograms and S is an example of a formal object.

Here is a version of the test application making use of the generic Perform_Test procedure:

Listing 38: generic_reverse.ads

```

1 generic
2   type T is private;
3   type Index is range <>;
4   type Array_T is
5     array (Index range <>) of T;
6 procedure Generic_Reverse (X : in out Array_T);

```

Listing 39: generic_reverse.adb

```

1 procedure Generic_Reverse (X : in out Array_T) is
2 begin
3   for I in X'First .. (X'Last + X'First) / 2 loop
4     declare
5       Tmp      : T;
6       X_Left  : T
7         renames X (I);
8       X_Right : T
9         renames X (X'Last + X'First - I);
10    begin
11      Tmp := X_Left;
12      X_Left := X_Right;
13      X_Right := Tmp;
14    end;
15   end loop;
16 end Generic_Reverse;
17

```

Listing 40: perform_test.ads

```

1 generic
2   type T is private;
3   type Index is range <>;
4   type Array_T is
5     array (Index range <>) of T;
6   S : String;
7   with function Image (E : T) return String is <>;
8   with procedure Test (X : in out Array_T);
9 procedure Perform_Test (X : in out Array_T);

```

Listing 41: perform_test.adb

```

1 with Ada.Text_IO;  use Ada.Text_IO;
2
3 procedure Perform_Test (X : in out Array_T) is
4 begin
5   for C of X loop
6     Put_Line (S & ": " & Image (C));
7   end loop;
8
9   New_Line;
10  Put_Line ("Testing " & S & "...");
11  New_Line;
12  Test (X);
13
14  for C of X loop

```

(continues on next page)

(continued from previous page)

```
15     Put_Line (S & ": " & Image (C));
16   end loop;
17 end Perform_Test;
```

Listing 42: colors.ads

```
1 with Generic_Reverse;
2
3 package Colors is
4
5   type Color is (Black, Red, Green,
6                 Blue, White);
7
8   type Color_Array is
9     array (Integer range <>) of Color;
10
11  procedure Reverse_It is new
12    Generic_Reverse (T      => Color,
13                     Index  => Integer,
14                     Array_T => Color_Array);
15
16 end Colors;
```

Listing 43: test_reverse_colors.adb

```
1 with Colors;      use Colors;
2 with Perform_Test;
3
4 procedure Test_Reverse_Colors is
5
6   procedure Perform_Test_Reverse_It is new
7     Perform_Test (T      => Color,
8                   Index  => Integer,
9                   Array_T => Color_Array,
10                  S      => "My_Color",
11                  Image  => Color'Image,
12                  Test   => Reverse_It);
13
14   My_Colors : Color_Array (1 .. 5) :=
15     (Black, Red, Green, Blue, White);
16
17 begin
18   Perform_Test_Reverse_It (My_Colors);
19 end Test_Reverse_Colors;
```

Runtime output

```
My_Color: BLACK
My_Color: RED
My_Color: GREEN
My_Color: BLUE
My_Color: WHITE

Testing My_Color...

My_Color: WHITE
My_Color: BLUE
My_Color: GREEN
My_Color: RED
My_Color: BLACK
```

In this example, we create the procedure `Perform_Test_Reverse_It` as an instance of the generic procedure (`Perform_Test`). Note that:

- For the formal `Image` function, we use the '`Image`' attribute of the `Color` type
- For the formal `Test` procedure, we reference the `Reverse_Array` procedure from the package.

EXCEPTIONS

Ada uses exceptions for error handling. Unlike many other languages, Ada speaks about *raising*, not *throwing*, an exception and *handling*, not *catching*, an exception.

14.1 Exception declaration

Ada exceptions are not types, but instead objects, which may be peculiar to you if you're used to the way Java or Python support exceptions. Here's how you declare an exception:

Listing 1: exceptions.ads

```
1 package Exceptions is
2     My_Except : exception;
3     -- Like an object. *NOT* a type !
4 end Exceptions;
```

Even though they're objects, you're going to use each declared exception object as a "kind" or "family" of exceptions. Ada does not require that a subprogram declare every exception it can potentially raise.

14.2 Raising an exception

To raise an exception of our newly declared exception kind, do the following:

Listing 2: main.adb

```
1 with Exceptions; use Exceptions;
2
3 procedure Main is
4 begin
5     raise My_Except;
6     -- Execution of current control flow
7     -- abandoned; an exception of kind
8     -- "My_Except" will bubble up until it
9     -- is caught.
10
11    raise My_Except with "My exception message";
12    -- Execution of current control flow
13    -- abandoned; an exception of kind
14    -- "My_Except" with associated string will
15    -- bubble up until it is caught.
16 end Main;
```

Build output

```
main.adb:11:04: warning: unreachable code [enabled by default]
```

Runtime output

```
raised EXCEPTIONS.MY_EXCEPT : main.adb:5
```

14.3 Handling an exception

Next, we address how to handle exceptions that were raised by us or libraries that we call. The neat thing in Ada is that you can add an exception handler to any statement block as follows:

Listing 3: open_file.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Exceptions;  use Ada.Exceptions;
3
4  procedure Open_File is
5    File : File_Type;
6  begin
7    -- Block (sequence of statements)
8    begin
9      Open (File, In_File, "input.txt");
10   exception
11     when E : Name_Error =>
12       -- ^ Exception to be handled
13       Put ("Cannot open input file : ");
14       Put_Line (Exception_Message (E));
15       raise;
16       -- Reraise current occurrence
17   end;
18 end Open_File;
```

Runtime output

```
Cannot open input file : input.txt: No such file or directory
raised ADA.IO_EXCEPTIONS.NAME_ERROR : input.txt: No such file or directory
```

In the example above, we're using the `Exception_Message` function from the `Ada.Exceptions` package. This function returns the message associated with the exception as a string.

You don't need to introduce a block just to handle an exception: you can add it to the statements block of your current subprogram:

Listing 4: open_file.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Exceptions;  use Ada.Exceptions;
3
4  procedure Open_File is
5    File : File_Type;
6  begin
7    Open (File, In_File, "input.txt");
8    -- Exception block can be added to any block
9  exception
```

(continues on next page)

(continued from previous page)

```

10  when Name_Error =>
11    Put ("Cannot open input file");
12 end Open_File;
```

Build output

```
open_file.adb:2:09: warning: no entities of "Ada.Exceptions" are referenced [-gnatwu]
open_file.adb:2:23: warning: use clause for package "Exceptions" has no effect [-gnatwu]
```

Runtime output

```
Cannot open input file
```

Attention

Exception handlers have an important restriction that you need to be careful about: Exceptions raised in the declarative section are not caught by the handlers of that block. So for example, in the following code, the exception will not be caught.

Listing 5: be_careful.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Exceptions;  use Ada.Exceptions;
3
4  procedure Be_Careful is
5    function Dangerous return Integer is
6    begin
7      raise Constraint_Error;
8      return 42;
9    end Dangerous;
10
11 begin
12   declare
13     A : Integer := Dangerous;
14   begin
15     Put_Line (Integer'Image (A));
16   exception
17     when Constraint_Error =>
18       Put_Line ("error!");
19   end;
20 end Be_Careful;
```

Build output

```
be_careful.adb:2:09: warning: no entities of "Ada.Exceptions" are referenced [-gnatwu]
be_careful.adb:2:23: warning: use clause for package "Exceptions" has no effect [-gnatwu]
be_careful.adb:13:07: warning: "A" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
raised CONSTRAINT_ERROR : be_careful.adb:7 explicit raise
```

This is also the case for the top-level exception block that is part of the current subprogram.

14.4 Predefined exceptions

Ada has a very small number of predefined exceptions:

- `Constraint_Error` is the main one you might see. It's raised:
 - When bounds don't match or, in general, any violation of constraints.
 - In case of overflow
 - In case of null dereferences
 - In case of division by 0
- `Program_Error` might appear, but probably less often. It's raised in more arcane situations, such as for order of elaboration issues and some cases of detectable erroneous execution.
- `Storage_Error` will happen because of memory issues, such as:
 - Not enough memory (allocator)
 - Not enough stack
- `Tasking_Error` will happen with task related errors, such as any error happening during task activation.

You should not reuse predefined exceptions. If you do then, it won't be obvious when one is raised that it is because something went wrong in a built-in language operation.

TASKING

Tasks and protected objects allow the implementation of concurrency in Ada. The following sections explain these concepts in more detail.

15.1 Tasks

A task can be thought as an application that runs *concurrently* with the main application. In other programming languages, a task might be called a [thread¹⁷](#), and tasking might be called [multithreading¹⁸](#).

Tasks may synchronize with the main application but may also process information completely independently from the main application. Here we show how this is accomplished.

15.1.1 Simple task

Tasks are declared using the keyword **task**. The task implementation is specified in a **task body** block. For example:

Listing 1: show_simple_task.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task is
4     task T;
5
6     task body T is
7         begin
8             Put_Line ("In task T");
9         end T;
10    begin
11        Put_Line ("In main");
12    end Show_Simple_Task;
```

Runtime output

```
In task T
In main
```

Here, we're declaring and implementing the task T. As soon as the main application starts, task T starts automatically — it's not necessary to manually start this task. By running the application above, we can see that both calls to Put_Line are performed.

Note that:

¹⁷ [https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))

¹⁸ [https://en.wikipedia.org/wiki/Thread_\(computing\)#Multithreading](https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading)

- The main application is itself a task (the main or “environment” task).
 - In this example, the subprogram Show_Simple_Task is the main task of the application.
- Task T is a subtask.
 - Each subtask has a master, which represents the program construct in which the subtask is declared. In this case, the main subprogram Show_Simple_Task is T's master.
 - The master construct is executed by some enclosing task, which we will refer to as the “master task” of the subtask.
- The number of tasks is not limited to one: we could include a task T2 in the example above.
 - This task also starts automatically and runs *concurrently* with both task T and the main task. For example:

Listing 2: show_simple_tasks.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Simple_Tasks is
4      task T;
5      task T2;
6
7      task body T is
8          begin
9              Put_Line ("In task T");
10             end T;
11
12      task body T2 is
13          begin
14              Put_Line ("In task T2");
15             end T2;
16
17      begin
18          Put_Line ("In main");
19      end Show_Simple_Tasks;
```

Runtime output

```
In task T
In task T2
In main
```

15.1.2 Simple synchronization

As we've just seen, as soon as the master construct reaches its “begin”, its subtasks also start automatically. The master continues its processing until it has nothing more to do. At that point, however, it will not terminate. Instead, the master waits until its subtasks have finished before it allows itself to complete. In other words, this waiting process provides synchronization between the master task and its subtasks. After this synchronization, the master construct will complete. For example:

Listing 3: show_simple_sync.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
```

(continues on next page)

(continued from previous page)

```

3  procedure Show_Simple_Sync is
4      task T;
5      task body T is
6          begin
7              for I in 1 .. 10 loop
8                  Put_Line ("hello");
9                  end loop;
10             end T;
11         begin
12             null;
13             -- Will wait here until all tasks
14             -- have terminated
15         end Show_Simple_Sync;

```

Runtime output

```

hello

```

The same mechanism is used for other subprograms that contain subtasks: the subprogram execution will wait for its subtasks to finish. So this mechanism is not limited to the main subprogram and also applies to any subprogram called by the main subprogram, directly or indirectly.

Synchronization also occurs if we move the task to a separate package. In the example below, we declare a task T in the package Simple_Sync_Pkg.

Listing 4: simple_sync_pkg.ads

```

1  package Simple_Sync_Pkg is
2      task T;
3  end Simple_Sync_Pkg;

```

This is the corresponding package body:

Listing 5: simple_sync_pkg.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Simple_Sync_Pkg is
4      task body T is
5          begin
6              for I in 1 .. 10 loop
7                  Put_Line ("hello");
8                  end loop;
9              end T;
10         end Simple_Sync_Pkg;

```

Because the package is **with**'ed by the main procedure, the task T defined in the package will become a subtask of the main task. For example:

Listing 6: test_simple_sync_pkg.adb

```
1 with Simple_Sync_Pkg;
2
3 procedure Test_Simple_Sync_Pkg is
4 begin
5   null;
6   -- Will wait here until all tasks
7   -- have terminated
8 end Test_Simple_Sync_Pkg;
```

Build output

```
test_simple_sync_pkg.adb:1:06: warning: unit "Simple_Sync_Pkg" is not referenced [-gnatwu]
```

Runtime output

```
hello
```

As soon as the main subprogram returns, the main task synchronizes with any subtasks spawned by packages T from Simple_Sync_Pkg before finally terminating.

15.1.3 Delay

We can introduce a delay by using the keyword **delay**. This puts the current task to sleep for the length of time (in seconds) specified in the delay statement. For example:

Listing 7: show_delay.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Delay is
4
5   task T;
6
7   task body T is
8   begin
9     for I in 1 .. 5 loop
10       Put_Line ("hello from task T");
11       delay 1.0;
12       -- ^ Wait 1.0 seconds
13     end loop;
14   end T;
15   begin
16     delay 1.5;
17     Put_Line ("hello from main");
18   end Show_Delay;
```

Runtime output

```
hello from task T
hello from task T
hello from main
hello from task T
hello from task T
hello from task T
```

In this example, we're making the task T wait one second after each time it displays the "hello" message. In addition, the main task is waiting 1.5 seconds before displaying its own "hello" message

15.1.4 Synchronization: rendezvous

The only type of synchronization we've seen so far is the one that happens automatically at the end of a master construct with a subtask. You can also define custom synchronization points using the keyword **entry**. An **entry** can be viewed as a special kind of subprogram, which is called by another task using a similar syntax, as we will see later.

In the task body definition, you define which part of the task will accept the entries by using the keyword **accept**. A task proceeds until it reaches an **accept** statement and then waits for some other task to synchronize with it. Specifically,

- The task with the entry waits at that point (in the **accept** statement), ready to accept a call to the corresponding entry from the master task.
- The other task calls the task entry, in a manner similar to a procedure call, to synchronize with the entry.

This synchronization between tasks is called a *rendezvous*. Let's see an example:

Listing 8: show_rendezvous.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Rendezvous is
4
5      task T is
6          entry Start;
7      end T;
8
9      task body T is
10         begin
11             accept Start;
12                 -- ^ Waiting for somebody
13                 -- to call the entry
14
15             Put_Line ("In T");
16         end T;
17
18     begin
19         Put_Line ("In Main");
20
21         -- Calling T's entry:
22         T.Start;
23     end Show_Rendezvous;
```

Runtime output

```
In Main
In T
```

In this example, we declare an entry Start for task T. In the task body, we implement this entry using `accept Start`. When task T reaches this point, it waits for some other task to call its entry. This synchronization occurs in the `T.Start` statement. After the rendezvous completes, the main task and task T again run concurrently until they synchronize one final time when the main subprogram `Show_Rendezvous` finishes.

An entry may be used to perform more than a simple task synchronization: it also may perform multiple statements during the time both tasks are synchronized. We do this with a `do ... end` block. For the previous example, we would simply write `accept Start do <statements>; end;`. We use this kind of block in the next example.

15.1.5 Select loop

There's no limit to the number of times an entry can be accepted. We could even create an infinite loop in the task and accept calls to the same entry over and over again. An infinite loop, however, prevents the subtask from finishing, so it blocks its master task when it reaches the end of its processing. Therefore, a loop containing `accept` statements in a task body can be used in conjunction with a `select ... or terminate` statement. In simple terms, this statement allows its master task to automatically terminate the subtask when the master construct reaches its end. For example:

Listing 9: `show_rendezvous_loop.adb`

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Rendezvous_Loop is
4
5      task T is
6          entry Reset;
7          entry Increment;
8      end T;
9
10     task body T is
11         Cnt : Integer := 0;
12     begin
13         loop
14             select
15                 accept Reset do
16                     Cnt := 0;
17                     end Reset;
18                     Put_Line ("Reset");
19                 or
20                 accept Increment do
21                     Cnt := Cnt + 1;
22                     end Increment;
23                     Put_Line ("In T's loop (" & Integer'Image (Cnt) & ")");
24                 or
25                     terminate;
26                     end select;
27             end loop;
28         end T;
29
30     begin
31         Put_Line ("In Main");
32
33         for I in 1 .. 4 loop
34             -- Calling T's entry multiple times
35             T.Increment;
36
37

```

(continues on next page)

(continued from previous page)

```

38   end loop;
39
40   T.Reset;
41   for I in 1 .. 4 loop
42     -- Calling T's entry multiple times
43     T.Increment;
44   end loop;
45
46 end Show_Rendezvous_Loop;

```

Runtime output

```

In Main
In T's loop ( 1)
In T's loop ( 2)
In T's loop ( 3)
In T's loop ( 4)
Reset
In T's loop ( 1)
In T's loop ( 2)
In T's loop ( 3)
In T's loop ( 4)

```

In this example, the task body implements an infinite loop that accepts calls to the Reset and Increment entry. We make the following observations:

- The `accept E do ... end` block is used to increment a counter.
 - As long as task T is performing the `do ... end` block, the main task waits for the block to complete.
- The main task is calling the Increment entry multiple times in the loop from `1 .. 4`. It is also calling the Reset entry before the second loop.
 - Because task T contains an infinite loop, it always accepts calls to the Reset and Increment entries.
 - When the master construct of the subtask (the `Show_Rendezvous_Loop` subprogram) completes, it checks the status of the T task. Even though task T could accept new calls to the Reset or Increment entries, the master construct is allowed to terminate task T due to the `or terminate` part of the `select` statement.

15.1.6 Cycling tasks

In a previous example, we saw how to delay a task a specified time by using the `delay` keyword. However, using delay statements in a loop is not enough to guarantee regular intervals between those delay statements. For example, we may have a call to a computationally intensive procedure between executions of successive delay statements:

```

while True loop
  delay 1.0;
  -- ^ Wait 1.0 seconds
  Computational_Intensive_App;
end loop;

```

In this case, we can't guarantee that exactly 10 seconds have elapsed after 10 calls to the `delay` statement because a time drift may be introduced by the `Computational_Intensive_App` procedure. In many cases, this time drift is not relevant, so using the `delay` keyword is good enough.

However, there are situations where a time drift isn't acceptable. In those cases, we need to use the `delay until` statement, which accepts a precise time for the end of the delay, allowing us to define a regular interval. This is useful, for example, in real-time applications.

We will soon see an example of how this time drift may be introduced and how the `delay until` statement circumvents the problem. But before we do that, we look at a package containing a procedure allowing us to measure the elapsed time (`Show_Elapsed_Time`) and a dummy `Computational_Intensive_App` procedure which is simulated by using a simple delay. This is the complete package:

Listing 10: `delay_aux_pkg.ads`

```
1  with Ada.Real_Time; use Ada.Real_Time;
2
3  package Delay_Aux_Pkg is
4
5      function Get_Start_Time return Time
6          with Inline;
7
8      procedure Show_Elapsed_Time
9          with Inline;
10
11     procedure Computational_Intensive_App;
12
13     Start_Time    : Time := Clock;
14
15     function Get_Start_Time return Time is (Start_Time);
16
17 end Delay_Aux_Pkg;
```

Listing 11: `delay_aux_pkg.adb`

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Delay_Aux_Pkg is
4
5      procedure Show_Elapsed_Time is
6          Now_Time      : Time;
7          Elapsed_Time : Time_Span;
8
9      begin
10         Now_Time      := Clock;
11         Elapsed_Time := Now_Time - Start_Time;
12         Put_Line ("Elapsed time "
13                    & Duration'Image (To_Duration (Elapsed_Time))
14                    & " seconds");
15     end Show_Elapsed_Time;
16
17     procedure Computational_Intensive_App is
18     begin
19         delay 0.5;
20     end Computational_Intensive_App;
21
22 end Delay_Aux_Pkg;
```

Using this auxiliary package, we're now ready to write our time-drifting application:

Listing 12: `show_time_task.adb`

```
1  with Ada.Text_IO;   use Ada.Text_IO;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  with Delay_Aux_Pkg;
```

(continues on next page)

(continued from previous page)

```

5  procedure Show_Time_Task is
6    package Aux renames Delay_Aux_Pkg;
7
8    task T;
9
10   task body T is
11     Cnt : Integer := 1;
12   begin
13     for I in 1 .. 5 loop
14       delay 1.0;
15
16       Aux.Show_Elapsed_Time;
17       Aux.Computational_Intensive_App;
18
19       Put_Line ("Cycle # "
20                 & Integer'Image (Cnt));
21       Cnt := Cnt + 1;
22     end loop;
23     Put_Line ("Finished time-drifting loop");
24   end T;
25
26 begin
27   null;
28 end Show_Time_Task;

```

Build output

```

show_time_task.adb:2:09: warning: no entities of "Ada.Real_Time" are referenced [-
→gnatwu]
show_time_task.adb:2:21: warning: use clause for package "Real_Time" has no effect →
→[-gnatwu]

```

Runtime output

```

Elapsed time 1.000455399 seconds
Cycle # 1
Elapsed time 2.501236592 seconds
Cycle # 2
Elapsed time 4.001556441 seconds
Cycle # 3
Elapsed time 5.502125295 seconds
Cycle # 4
Elapsed time 7.002507780 seconds
Cycle # 5
Finished time-drifting loop

```

We can see by running the application that we already have a time difference of about four seconds after three iterations of the loop due to the drift introduced by Computational_Intensive_App. Using the **delay until** statement, however, we're able to avoid this time drift and have a regular interval of exactly one second:

Listing 13: show_time_task.adb

```

1  with Ada.Text_IO;  use Ada.Text_IO;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  with Delay_Aux_Pkg;
5
6  procedure Show_Time_Task is
7    package Aux renames Delay_Aux_Pkg;

```

(continues on next page)

(continued from previous page)

```
8   task T;
9
10  task body T is
11    Cycle : constant Time_Span :=
12      Milliseconds (1000);
13    Next  : Time := Aux.Get_Start_Time
14      + Cycle;
15
16    Cnt   : Integer := 1;
17  begin
18    for I in 1 .. 5 loop
19      delay until Next;
20
21      Aux.Show_Elapsed_Time;
22      Aux.Computational_Intensive_App;
23
24      -- Calculate next execution time
25      -- using a cycle of one second
26      Next := Next + Cycle;
27
28      Put_Line ("Cycle # "
29                  & Integer'Image (Cnt));
30      Cnt := Cnt + 1;
31    end loop;
32    Put_Line ("Finished cycling");
33  end T;
34
35 begin
36  null;
37 end Show_Time_Task;
```

Runtime output

```
Elapsed time 1.000176310 seconds
Cycle # 1
Elapsed time 2.000125986 seconds
Cycle # 2
Elapsed time 3.000232581 seconds
Cycle # 3
Elapsed time 4.000131149 seconds
Cycle # 4
Elapsed time 5.000215936 seconds
Cycle # 5
Finished cycling
```

Now, as we can see by running the application, the **delay until** statement ensures that the Computational_Intensive_App doesn't disturb the regular interval of one second between iterations.

15.2 Protected objects

When multiple tasks are accessing shared data, corruption of that data may occur. For example, data may be inconsistent if one task overwrites parts of the information that's being read by another task at the same time. In order to avoid these kinds of problems and ensure information is accessed in a coordinated way, we use *protected objects*.

Protected objects encapsulate data and provide access to that data by means of *protected operations*, which may be subprograms or protected entries. Using protected objects ensures that data is not corrupted by race conditions or other concurrent access.

Important

Objects can be protected from concurrent access using Ada tasks. In fact, this was the *only* way of protecting objects from concurrent access in Ada 83 (the first version of the Ada language). However, the use of protected objects is much simpler than using similar mechanisms implemented using only tasks. Therefore, you should use protected objects when your main goal is only to protect data.

15.2.1 Simple object

You declare a protected object with the **protected** keyword. The syntax is similar to that used for packages: you can declare operations (e.g., procedures and functions) in the public part and data in the private part. The corresponding implementation of the operations is included in the **protected body** of the object. For example:

Listing 14: show_protected_objects.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Protected_Objects is
4
5      protected Obj is
6          -- Operations go here (only subprograms)
7          procedure Set (V : Integer);
8          function Get return Integer;
9      private
10         -- Data goes here
11         Local : Integer := 0;
12     end Obj;
13
14     protected body Obj is
15         -- procedures can modify the data
16         procedure Set (V : Integer) is
17             begin
18                 Local := V;
19             end Set;
20
21         -- functions cannot modify the data
22         function Get return Integer is
23             begin
24                 return Local;
25             end Get;
26     end Obj;
27
28 begin
29     Obj.Set (5);

```

(continues on next page)

(continued from previous page)

```

30 Put_Line ("Number is: "
31   & Integer'Image (Obj.Get));
32 end Show_Protected_Objects;

```

Runtime output

```
Number is: 5
```

In this example, we define two operations for Obj: Set and Get. The implementation of these operations is in the Obj body. The syntax used for writing these operations is the same as that for normal procedures and functions. The implementation of protected objects is straightforward — we simply access and update Local in these subprograms. To call these operations in the main application, we use prefixed notation, e.g., Obj.Get.

15.2.2 Entries

In addition to protected procedures and functions, you can also define protected entry points. Do this using the `entry` keyword. Protected entry points allow you to define barriers using the `when` keyword. Barriers are conditions that must be fulfilled before the entry can start performing its actual processing — we speak of *releasing* the barrier when the condition is fulfilled.

The previous example used procedures and functions to define operations on the protected objects. However, doing so permits reading protected information (via Obj.Get) before it's set (via Obj.Set). To allow that to be a defined operation, we specified a default value (0). Instead, by rewriting Obj.Get using an `entry` instead of a function, we implement a barrier, ensuring no task can read the information before it's been set.

The following example implements the barrier for the Obj.Get operation. It also contains two concurrent subprograms (main task and task T) that try to access the protected object.

Listing 15: show_protected_objects_entries.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Protected_ObjectsEntries is
4
5   protected Obj is
6     procedure Set (V : Integer);
7     entry Get (V : out Integer);
8   private
9     Local : Integer;
10    Is_Set : Boolean := False;
11  end Obj;
12
13  protected body Obj is
14    procedure Set (V : Integer) is
15      begin
16        Local := V;
17        Is_Set := True;
18      end Set;
19
20    entry Get (V : out Integer)
21      when Is_Set is
22      -- Entry is blocked until the
23      -- condition is true. The barrier
24      -- is evaluated at call of entries
25      -- and at exits of procedures and
26      -- entries. The calling task sleeps

```

(continues on next page)

(continued from previous page)

```

27      -- until the barrier is released.
28  begin
29    V := Local;
30    Is_Set := False;
31  end Get;
32 end Obj;
33
34 N : Integer := 0;
35
36 task T;
37
38 task body T is
39 begin
40   Put_Line ("Task T will delay for 4 seconds...");
41   delay 4.0;
42   Put_Line ("Task T will set Obj...");
43   Obj.Set (5);
44   Put_Line ("Task T has just set Obj...");
45 end T;
46 begin
47   Put_Line ("Main application will get Obj...");
48   Obj.Get (N);
49   Put_Line ("Main application has just retrieved Obj...");
50   Put_Line ("Number is: " & Integer'Image (N));
51
52 end Show_Protected_Objects_Entries;

```

Runtime output

```

Task T will delay for 4 seconds...
Main application will get Obj...
Task T will set Obj...
Task T has just set Obj...
Main application has just retrieved Obj...
Number is: 5

```

As we see by running it, the main application waits until the protected object is set (by the call to `Obj.Set` in task T) before it reads the information (via `Obj.Get`). Because a 4-second delay has been added in task T, the main application is also delayed by 4 seconds. Only after this delay does task T set the object and release the barrier in `Obj.Get` so that the main application can then resume processing (after the information is retrieved from the protected object).

15.3 Task and protected types

In the previous examples, we defined single tasks and protected objects. We can, however, generalize tasks and protected objects using type definitions. This allows us, for example, to create multiple tasks based on just a single task type.

15.3.1 Task types

A task type is a generalization of a task. The declaration is similar to simple tasks: you replace `task` with `task type`. The difference between simple tasks and task types is that task types don't create actual tasks that automatically start. Instead, a task object declaration is needed. This is exactly the way normal variables and types work: objects are only created by variable definitions, not type definitions.

To illustrate this, we repeat our first example:

Listing 16: show_simple_task.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task is
4     task T;
5
6     task body T is
7         begin
8             Put_Line ("In task T");
9         end T;
10    begin
11        Put_Line ("In main");
12    end Show_Simple_Task;
```

Runtime output

```
In task T
In main
```

We now rewrite it by replacing `task T` with `task type TT`. We declare a task (`A_Task`) based on the task type `TT` after its definition:

Listing 17: show_simple_task_type.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task_Type is
4     task type TT;
5
6     task body TT is
7         begin
8             Put_Line ("In task type TT");
9         end TT;
10
11    A_Task : TT;
12    begin
13        Put_Line ("In main");
14    end Show_Simple_Task_Type;
```

Runtime output

```
In task type TT
In main
```

We can extend this example and create an array of tasks. Since we're using the same syntax as for variable declarations, we use a similar syntax for task types: `array (<>) of Task_Type`. Also, we can pass information to the individual tasks by defining a `Start` entry. Here's the updated example:

Listing 18: show_task_type_array.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Task_Type_Array is
4      task type TT is
5          entry Start (N : Integer);
6      end TT;
7
8      task body TT is
9          Task_N : Integer;
10     begin
11         accept Start (N : Integer) do
12             Task_N := N;
13         end Start;
14         Put_Line ("In task T: "
15                   & Integer'Image (Task_N));
16     end TT;
17
18     My_Tasks : array (1 .. 5) of TT;
19 begin
20     Put_Line ("In main");
21
22     for I in My_Tasks'Range loop
23         My_Tasks (I).Start (I);
24     end loop;
25 end Show_Task_Type_Array;

```

Runtime output

```

In main
In task T: 1
In task T: 2
In task T: 3
In task T: 4
In task T: 5

```

In this example, we're declaring five tasks in the array `My_Tasks`. We pass the array index to the individual tasks in the entry point (`Start`). After the synchronization between the individual subtasks and the main task, each subtask calls `Put_Line` concurrently.

15.3.2 Protected types

A protected type is a generalization of a protected object. The declaration is similar to that for protected objects: you replace `protected` with `protected type`. Like task types, protected types require an object declaration to create actual objects. Again, this is similar to variable declarations and allows for creating arrays (or other composite objects) of protected objects.

We can reuse a previous example and rewrite it to use a protected type:

Listing 19: show_protected_object_type.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Protected_Object_Type is
4
5      protected type P_Obj_Type is
6          procedure Set (V : Integer);
7          function Get return Integer;

```

(continues on next page)

(continued from previous page)

```
8      private
9          Local : Integer := 0;
10         end P_Obj_Type;
11
12     protected body P_Obj_Type is
13         procedure Set (V : Integer) is
14             begin
15                 Local := V;
16             end Set;
17
18         function Get return Integer is
19             begin
20                 return Local;
21             end Get;
22         end P_Obj_Type;
23
24     Obj : P_Obj_Type;
25     begin
26         Obj.Set (5);
27         Put_Line ("Number is: "
28                     & Integer'Image (Obj.Get));
29     end Show_Protected_Object_Type;
```

Runtime output

```
Number is: 5
```

In this example, instead of directly defining the protected object Obj, we first define a protected type P_Obj_Type and then declare Obj as an object of that protected type. Note that the main application hasn't changed: we still use Obj.Set and Obj.Get to access the protected object, just like in the original example.

DESIGN BY CONTRACTS

Contracts are used in programming to codify expectations. Parameter modes of a subprogram can be viewed as a simple form of contracts. When the specification of subprogram `Op` declares a parameter using `in` mode, the caller of `Op` knows that the `in` argument won't be changed by `Op`. In other words, the caller expects that `Op` doesn't modify the argument it's providing, but just reads the information stored in the argument. Constraints and subtypes are other examples of contracts. In general, these specifications improve the consistency of the application.

Design-by-contract programming refers to techniques that include pre- and postconditions, subtype predicates, and type invariants. We study those topics in this chapter.

16.1 Pre- and postconditions

Pre- and postconditions provide expectations regarding input and output parameters of subprograms and return value of functions. If we say that certain requirements must be met before calling a subprogram `Op`, those are preconditions. Similarly, if certain requirements must be met after a call to the subprogram `Op`, those are postconditions. We can think of preconditions and postconditions as promises between the subprogram caller and the callee: a precondition is a promise from the caller to the callee, and a postcondition is a promise in the other direction.

Pre- and postconditions are specified using an aspect clause in the subprogram declaration. A `with Pre => <condition>` clause specifies a precondition and a `with Post => <condition>` clause specifies a postcondition.

The following code shows an example of preconditions:

Listing 1: show_simple_precondition.adb

```
1  procedure Show_Simple_Precondition is
2
3      procedure DB_Entry (Name : String;
4                          Age  : Natural)
5          with Pre => Name'Length > 0
6      is
7          begin
8              -- Missing implementation
9              null;
10             end DB_Entry;
11         begin
12             DB_Entry ("John", 30);
13
14             -- Precondition will fail!
15             DB_Entry ("", 21);
16         end Show_Simple_Precondition;
```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : failed precondition from show_simple_
↳precondition.adb:5
```

In this example, we want to prevent the name field in our database from containing an empty string. We implement this requirement by using a precondition requiring that the length of the string used for the Name parameter of the DB_Entry procedure is greater than zero. If the DB_Entry procedure is called with an empty string for the Name parameter, the call will fail because the precondition is not met.

In the GNAT toolchain

GNAT handles pre- and postconditions by generating runtime assertions for them. By default, however, assertions aren't enabled. Therefore, in order to check pre- and postconditions at runtime, you need to enable assertions by using the `-gnata` switch.

Before we get to our next example, let's briefly discuss quantified expressions, which are quite useful in concisely writing pre- and postconditions. Quantified expressions return a Boolean value indicating whether elements of an array or container match the expected condition. They have the form: `(for all I in A'Range => <condition on A(I)>)`, where A is an array and I is an index. Quantified expressions using `for all` check whether the condition is true for every element. For example:

```
(for all I in A'Range => A (I) = 0)
```

This quantified expression is only true when all elements of the array A have a value of zero.

Another kind of quantified expressions uses `for some`. The form looks similar: `(for some I in A'Range => <condition on A(I)>)`. However, in this case the qualified expression tests whether the condition is true only on *some* elements (hence the name) instead of all elements.

We illustrate postconditions using the following example:

Listing 2: show_simple_postcondition.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Simple_Postcondition is
4
5    type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;
6
7    type Int_8_Array is
8      array (Integer range <>) of Int_8;
9
10   function Square (A : Int_8) return Int_8 is
11     (A * A)
12     with Post => (if abs A in 0 | 1
13                   then Square'Result = abs A
14                   else Square'Result > A);
15
16   procedure Square (A : in out Int_8_Array)
17     with Post => (for all I in A'Range =>
18                     A (I) = A'Old (I) * A'Old (I))
19   is
20   begin
21     for V of A loop
22       V := Square (V);
23     end loop;
```

(continues on next page)

(continued from previous page)

```

24  end Square;
25
26  V : Int_8_Array := (-2, -1, 0, 1, 10, 11);
27 begin
28   for E of V loop
29     Put_Line ("Original: "
30               & Int_8'Image (E));
31   end loop;
32   New_Line;
33
34   Square (V);
35   for E of V loop
36     Put_Line ("Square:    "
37               & Int_8'Image (E));
38   end loop;
39 end Show_Simple_Postcondition;

```

Runtime output

```

Original: -2
Original: -1
Original: 0
Original: 1
Original: 10
Original: 11

Square:    4
Square:    1
Square:    0
Square:    1
Square: 100
Square: 121

```

We declare a signed 8-bit type `Int_8` and an array of that type (`Int_8_Array`). We want to ensure each element of the array is squared after calling the procedure `Square` for an object of the `Int_8_Array` type. We do this with a postcondition using a `for all` expression. This postcondition also uses the '`Old`' attribute to refer to the original value of the parameter (before the call).

We also want to ensure that the result of calls to the `Square` function for the `Int_8` type are greater than the input to that call. To do that, we write a postcondition using the '`Result`' attribute of the function and comparing it to the input value.

We can use both pre- and postconditions in the declaration of a single subprogram. For example:

Listing 3: `show_simple_contract.adb`

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Simple_Contract is
4
5    type Int_8 is range -2 ** 7 .. 2 ** 7 - 1;
6
7    function Square (A : Int_8) return Int_8 is
8      (A * A)
9      with
10        Pre  => (Integer'Size >= Int_8'Size * 2
11                  and Integer (A) * Integer (A) <=
12                      Integer (Int_8'Last)),
13        Post => (if abs A in 0 | 1

```

(continues on next page)

(continued from previous page)

```

14      then Square'Result = abs A
15      else Square'Result > A);
16
17  V : Int_8;
18 begin
19    V := Square (11);
20    Put_Line ("Square of 11 is " & Int_8'Image (V));
21
22    -- Precondition will fail...
23    V := Square (12);
24    Put_Line ("Square of 12 is " & Int_8'Image (V));
25 end Show_Simple_Contract;

```

Runtime output

```

Square of 11 is 121
raised ADA ASSERTIONS ASSERTION_ERROR : failed precondition from show_simple_
contract.adb:10

```

In this example, we want to ensure that the input value of calls to the `Square` function for the `Int_8` type won't cause overflow in that function. We do this by converting the input value to the `Integer` type, which is used for the temporary calculation, and check if the result is in the appropriate range for the `Int_8` type. We have the same postcondition in this example as in the previous one.

16.2 Predicates

Predicates specify expectations regarding types. They're similar to pre- and postconditions, but apply to types instead of subprograms. Their conditions are checked for each object of a given type, which allows verifying that an object of type T is conformant to the requirements of its type.

There are two kinds of predicates: static and dynamic. In simple terms, static predicates are used to check objects at compile-time, while dynamic predicates are used for checks at run time. Normally, static predicates are used for scalar types and dynamic predicates for the more complex types.

Static and dynamic predicates are specified using the following clauses, respectively:

- `with Static_Predicate => <property>`
- `with Dynamic_Predicate => <property>`

Let's use the following example to illustrate dynamic predicates:

Listing 4: `show_dynamic_predicate_courses.adb`

```

1  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2  with Ada.Calendar;           use Ada.Calendar;
3  with Ada.Containers.Vectors;
4
5  procedure Show_Dynamic_Predicate_Courses is
6
7    package Courses is
8      type Course_Container is private;
9
10   type Course is record
11     Name      : Unbounded_String;

```

(continues on next page)

(continued from previous page)

```

12      Start_Date : Time;
13      End_Date   : Time;
14  end record
15  with Dynamic_Predicate =>
16      Course.Start_Date <= Course.End_Date;
17
18  procedure Add (CC : in out Course_Container;
19                  C   :          Course);
20 private
21  package Course_Vectors is new
22      Ada.Containers.Vectors
23      (Index_Type    => Natural,
24       Element_Type  => Course);
25
26  type Course_Container is record
27      V : Course_Vectors.Vector;
28  end record;
29 end Courses;
30
31 package body Courses is
32  procedure Add (CC : in out Course_Container;
33                  C   :          Course) is
34  begin
35      CC.V.Append (C);
36  end Add;
37 end Courses;
38
39 use Courses;
40
41 CC : Course_Container;
42 begin
43     Add (CC,
44           Course'(
45               Name      => To_Unbounded_String
46                           ("Intro to Photography"),
47               Start_Date => Time_Of (2018, 5, 1),
48               End_Date   => Time_Of (2018, 5, 10)));
49
50 -- This should trigger an error in the
51 -- dynamic predicate check
52     Add (CC,
53           Course'(
54               Name      => To_Unbounded_String
55                           ("Intro to Video Recording"),
56               Start_Date => Time_Of (2019, 5, 1),
57               End_Date   => Time_Of (2018, 5, 10)));
58
59 end Show_Dynamic_Predicate_Courses;

```

Runtime output

```

raised ADA ASSERTIONS ASSERTION_ERROR : Dynamic_Predicate failed at show_dynamic_
→predicate_courses.adb:53

```

In this example, the package Courses defines a type Course and a type Course_Container, an object of which contains all courses. We want to ensure that the dates of each course are consistent, specifically that the start date is no later than the end date. To enforce this rule, we declare a dynamic predicate for the Course type that performs the check for each object. The predicate uses the type name where a variable of that type would normally be used: this is a reference to the instance of the object being tested.

Note that the example above makes use of unbounded strings and dates. Both types are available in Ada's standard library. Please refer to the following sections for more information about:

- the unbounded string type (Unbounded_String): [Unbounded Strings](#) (page 232) section;
- dates and times: [Dates & Times](#) (page 217) section.

Static predicates, as mentioned above, are mostly used for scalar types and checked during compilation. They're particularly useful for representing non-contiguous elements of an enumeration. A classic example is a list of week days:

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

We can easily create a sub-list of work days in the week by specifying a **subtype** with a range based on Week. For example:

```
subtype Work_Week is Week range Mon .. Fri;
```

Ranges in Ada can only be specified as contiguous lists: they don't allow us to pick specific days. However, we may want to create a list containing just the first, middle and last day of the work week. To do that, we use a static predicate:

```
subtype Check_Days is Work_Week
  with Static_Predicate => Check_Days in Mon | Wed | Fri;
```

Let's look at a complete example:

Listing 5: show_predicates.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Predicates is
4
5    type Week is (Mon, Tue, Wed, Thu,
6                  Fri, Sat, Sun);
7
8    subtype Work_Week is Week range Mon .. Fri;
9
10   subtype Test_Days is Work_Week
11     with Static_Predicate =>
12       Test_Days in Mon | Wed | Fri;
13
14   type Tests_Week is array (Week) of Natural
15   with Dynamic_Predicate =>
16     (for all I in Tests_Week'Range =>
17      (case I is
18        when Test_Days =>
19          Tests_Week (I) > 0,
20        when others      =>
21          Tests_Week (I) = 0));
22
23   Num_Tests : Tests_Week :=
24     (Mon => 3, Tue => 0,
25      Wed => 4, Thu => 0,
26      Fri => 2, Sat => 0,
27      Sun => 0);
28
29  procedure Display_Tests (N : Tests_Week) is
30  begin
31    for I in Test_Days loop
32      Put_Line ("# tests on "
```

(continues on next page)

(continued from previous page)

```

33      & Test_Days'Image (I)
34      & " => "
35      & Integer'Image (N (I)));
36  end loop;
37 end Display_Tests;

38
39 begin
40   Display_Tests (Num_Tests);
41
42   -- Assigning non-conformant values to
43   -- individual elements of the Tests_Week
44   -- type does not trigger a predicate
45   -- check:
46   Num_Tests (Tue) := 2;
47
48   -- However, assignments with the "complete"
49   -- Tests_Week type trigger a predicate
50   -- check. For example:
51
52   -- Num_Tests := (others => 0);
53
54   -- Also, calling any subprogram with
55   -- parameters of Tests_Week type
56   -- triggers a predicate check. Therefore,
57   -- the following line will fail:
58   Display_Tests (Num_Tests);
59 end Show_Predicates;

```

Runtime output

```

# tests on MON => 3
# tests on WED => 4
# tests on FRI => 2

raised ADA ASSERTIONS ASSERTION_ERROR : Dynamic_Predicate failed at show_
  predicates.adb:58

```

Here we have an application that wants to perform tests only on three days of the work week. These days are specified in the `Test_Days` subtype. We want to track the number of tests that occur each day. We declare the type `Tests_Week` as an array, an object of which will contain the number of tests done each day. According to our requirements, these tests should happen only in the aforementioned three days; on other days, no tests should be performed. This requirement is implemented with a dynamic predicate of the type `Tests_Week`. Finally, the actual information about these tests is stored in the array `Num_Tests`, which is an instance of the `Tests_Week` type.

The dynamic predicate of the `Tests_Week` type is verified during the initialization of `Num_Tests`. If we have a non-conformant value there, the check will fail. However, as we can see in our example, individual assignments to elements of the array do not trigger a check. We can't check for consistency at this point because the initialization of the a complex data structure (such as arrays or records) may not be performed with a single assignment. However, as soon as the object is passed as an argument to a subprogram, the dynamic predicate is checked because the subprogram requires the object to be consistent. This happens in the last call to `Display_Tests` in our example. Here, the predicate check fails because the previous assignment has a non-conformant value.

16.3 Type invariants

Type invariants are another way of specifying expectations regarding types. While predicates are used for *non-private* types, type invariants are used exclusively to define expectations about private types. If a type T from a package P has a type invariant, the results of operations on objects of type T are always consistent with that invariant.

Type invariants are specified with a **with** Type_Invariant => <property> clause. Like predicates, the *property* defines a condition that allows us to check if an object of type T is conformant to its requirements. In this sense, type invariants can be viewed as a sort of predicate for private types. However, there are some differences in terms of checks. The following table summarizes the differences:

Element	Subprogram parameter checks	Assignment checks
Predicates	On all in and out parameters	On assignments and explicit initializations
Type invariants	On out parameters returned from subprograms declared in the same public scope	On all initializations

We could rewrite our previous example and replace dynamic predicates by type invariants. It would look like this:

Listing 6: show_type_invariant.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
3  with Ada.Calendar;         use Ada.Calendar;
4  with Ada.Containers.Vectors;
5
6  procedure Show_Type_Invariant is
7
8      package Courses is
9          type Course is private
10         with Type_Invariant => Check (Course);
11
12         type Course_Container is private;
13
14         procedure Add (CC : in out Course_Container;
15                         C : Course);
16
17         function Init
18             (Name : String;
19              Start_Date, End_Date : Time) return Course;
20
21         function Check (C : Course) return Boolean;
22
23     private
24         type Course is record
25             Name : Unbounded_String;
26             Start_Date : Time;
27             End_Date : Time;
28         end record;
29
30         function Check (C : Course) return Boolean is
31             (C.Start_Date <= C.End_Date);
32
33     package Course_Vectors is new
34         Ada.Containers.Vectors
35             (Index_Type => Natural,
36              Element_Type => Course);

```

(continues on next page)

(continued from previous page)

```

37
38      type Course_Container is record
39          V : Course_Vectors.Vector;
40      end record;
41  end Courses;
42
43 package body Courses is
44     procedure Add (CC : in out Course_Container;
45                     C : Course) is
46 begin
47     CC.V.Append (C);
48 end Add;
49
50     function Init
51         (Name           : String;
52          Start_Date, End_Date : Time) return Course is
53 begin
54     return
55         Course'(Name        => To_Unbounded_String (Name),
56                  Start_Date => Start_Date,
57                  End_Date   => End_Date);
58 end Init;
59 end Courses;
60
61 use Courses;
62
63 CC : Course_Container;
64 begin
65     Add (CC,
66           Init (Name      => "Intro to Photography",
67                  Start_Date => Time_Of (2018, 5, 1),
68                  End_Date   => Time_Of (2018, 5, 10)));
69
70     -- This should trigger an error in the
71     -- type-invariant check
72     Add (CC,
73           Init (Name      => "Intro to Video Recording",
74                  Start_Date => Time_Of (2019, 5, 1),
75                  End_Date   => Time_Of (2018, 5, 10)));
76 end Show_Type_Invariant;

```

Build output

```

show_typeInvariant.adb:1:09: warning: no entities of "Ada.Text_IO" are referenced ↴ [-gnatwu]
show_typeInvariant.adb:1:29: warning: use clause for package "Text_IO" has no ↴ effect [-gnatwu]

```

Runtime output

```

raised ADA ASSERTIONS ASSERTION_ERROR : failed invariant from show_typeInvariant.
ADB:10

```

The major difference is that the Course type was a visible (public) type of the Courses package in the previous example, but in this example is a private type.

CHAPTER
SEVENTEEN

INTERFACING WITH C

Ada allows us to interface with code in many languages, including C and C++. This section discusses how to interface with C.

17.1 Multi-language project

By default, when using **gprbuild** we only compile Ada source files. To compile C files as well, we need to modify the project file used by **gprbuild**. We use the Languages entry, as in the following example:

```
project Multilang is
    for Languages use ("ada", "c");
    for Source_Dirs use ("src");
    for Main use ("main.adb");
    for Object_Dir use "obj";
end Multilang;
```

17.2 Type convention

To interface with data types declared in a C application, you specify the Convention aspect on the corresponding Ada type declaration. In the following example, we interface with the C_Enum enumeration declared in a C source file:

Listing 1: show_c_enum.adb

```
1 procedure Show_C_Enum is
2
3     type C_Enum is (A, B, C)
4         with Convention => C;
5         -- Use C convention for C_Enum
6 begin
7     null;
8 end Show_C_Enum;
```

To interface with C's built-in types, we use the `Interfaces.C` package, which contains most of the type definitions we need. For example:

Listing 2: show_c_struct.adb

```

1  with Interfaces.C; use Interfaces.C;
2
3  procedure Show_C_Struct is
4
5    type c_struct is record
6      a : int;
7      b : long;
8      c : unsigned;
9      d : double;
10   end record;
11   with Convention => C;
12
13 begin
14   null;
15 end Show_C_Struct;
```

Here, we're interfacing with a C struct (C_Struct) and using the corresponding data types in C (**int**, **long**, **unsigned** and **double**). This is the declaration in C:

Listing 3: c_struct.h

```

1 struct c_struct
2 {
3     int         a;
4     long        b;
5     unsigned    c;
6     double      d;
7 };
```

17.3 Foreign subprograms

17.3.1 Calling C subprograms in Ada

We use a similar approach when interfacing with subprograms written in C. Consider the following declaration in the C header file:

Listing 4: my_func.h

```

1 int my_func (int a);
```

Here's the corresponding C definition:

Listing 5: my_func.c

```

1 #include "my_func.h"
2
3 int my_func (int a)
4 {
5     return a * 2;
6 }
```

We can interface this code in Ada using the `Import` aspect. For example:

Listing 6: show_c_func.adb

```

1  with Interfaces.C; use Interfaces.C;
2  with Ada.Text_Io;  use Ada.Text_Io;
3
4  procedure Show_C_Func is
5
6    function my_func (a : int) return int
7    with
8      Import          => True,
9      Convention     => C;
10
11   -- Imports function 'my_func' from C.
12   -- You can now call it from Ada.
13
14  V : int;
15 begin
16  V := my_func (2);
17  Put_Line ("Result is " & int'Image (V));
18 end Show_C_Func;

```

If you want, you can use a different subprogram name in the Ada code. For example, we could call the C function Get_Value:

Listing 7: show_c_func.adb

```

1  with Interfaces.C; use Interfaces.C;
2  with Ada.Text_Io;  use Ada.Text_Io;
3
4  procedure Show_C_Func is
5
6    function Get_Value (a : int) return int
7    with
8      Import          => True,
9      Convention     => C,
10     External_Name  => "my_func";
11
12   -- Imports function 'my_func' from C and
13   -- rename it to 'Get_Value'
14
15  V : int;
16 begin
17  V := Get_Value (2);
18  Put_Line ("Result is " & int'Image (V));
19 end Show_C_Func;

```

17.3.2 Calling Ada subprograms in C

You can also call Ada subprograms from C applications. You do this with the Export aspect. For example:

Listing 8: c_api.ads

```

1  with Interfaces.C; use Interfaces.C;
2
3  package C_API is
4
5    function My_Func (a : int) return int
6    with

```

(continues on next page)

(continued from previous page)

```

7   Export      => True,
8   Convention  => C,
9   External_Name => "my_func";
10
11 end C_API;

```

This is the corresponding body that implements that function:

Listing 9: c_api.adb

```

1 package body C_API is
2
3   function My_Func (a : int) return int is
4   begin
5     return a * 2;
6   end My_Func;
7
8 end C_API;

```

On the C side, we do the same as we would if the function were written in C: simply declare it using the `extern` keyword. For example:

Listing 10: main.c

```

1 #include <stdio.h>
2
3 extern int my_func (int a);
4
5 int main (int argc, char **argv) {
6
7   int v = my_func(2);
8
9   printf("Result is %d\n", v);
10
11  return 0;
12 }

```

17.4 Foreign variables

17.4.1 Using C global variables in Ada

To use global variables from C code, we use the same method as subprograms: we specify the Import and Convention aspects for each variable we want to import.

Let's reuse an example from the previous section. We'll add a global variable (`func_cnt`) to count the number of times the function (`my_func`) is called:

Listing 11: test.h

```

1 extern int func_cnt;
2
3 int my_func (int a);

```

The variable is declared in the C file and incremented in `my_func`:

Listing 12: test.c

```

1 #include "test.h"
2
3 int func_cnt = 0;
4
5 int my_func (int a)
6 {
7     func_cnt++;
8
9     return a * 2;
10}

```

In the Ada application, we just reference the foreign variable:

Listing 13: show_c_func.adb

```

1 with Interfaces.C; use Interfaces.C;
2 with Ada.Text_Io;  use Ada.Text_Io;
3
4 procedure Show_C_Func is
5
6     function my_func (a : int) return int
7         with
8             Import      => True,
9             Convention => C;
10
11    V : int;
12
13    func_cnt : int
14        with
15            Import      => True,
16            Convention => C;
17        -- We can access the func_cnt variable
18        -- from test.c
19
20 begin
21    V := my_func (1);
22    V := my_func (2);
23    V := my_func (3);
24    Put_Line ("Result is " & int'Image (V));
25
26    Put_Line ("Function was called "
27              & int'Image (func_cnt)
28              & " times");
29 end Show_C_Func;

```

As we see by running the application, the value of the counter is the number of times `my_func` was called.

We can use the `External_Name` aspect to give a different name for the variable in the Ada application in the same way we do for subprograms.

17.4.2 Using Ada variables in C

You can also use variables declared in Ada files in C applications. In the same way as we did for subprograms, you do this with the Export aspect.

Let's reuse a past example and add a counter, as in the previous example, but this time have the counter incremented in Ada code:

Listing 14: c_api.ads

```

1  with Interfaces.C; use Interfaces.C;
2
3  package C_API is
4
5      func_cnt : int := 0
6      with
7          Export    => True,
8          Convention => C;
9
10     function My_Func (a : int) return int
11     with
12         Export    => True,
13         Convention => C,
14         External_Name => "my_func";
15
16 end C_API;

```

The variable is then increment in My_Func:

Listing 15: c_api.adb

```

1  package body C_API is
2
3      function My_Func (a : int) return int is
4          begin
5              func_cnt := func_cnt + 1;
6              return a * 2;
7          end My_Func;
8
9 end C_API;

```

In the C application, we just need to declare the variable and use it:

Listing 16: main.c

```

1 #include <stdio.h>
2
3 extern int my_func (int a);
4
5 extern int func_cnt;
6
7 int main (int argc, char **argv) {
8
9     int v;
10
11     v = my_func(1);
12     v = my_func(2);
13     v = my_func(3);
14
15     printf("Result is %d\n", v);
16
17     printf("Function was called %d times\n", func_cnt);

```

(continues on next page)

(continued from previous page)

```

18     return 0;
19 }
20

```

Again, by running the application, we see that the value from the counter is the number of times that `my_func` was called.

17.5 Generating bindings

In the examples above, we manually added aspects to our Ada code to correspond to the C source-code we're interfacing with. This is called creating a *binding*. We can automate this process by using the *Ada spec dump* compiler option: `-fdump-ada-spec`. We illustrate this by revisiting our previous example.

This was our C header file:

Listing 17: `my_func.c`

```

1 extern int func_cnt;
2
3 int my_func (int a);

```

To create Ada bindings, we'll call the compiler like this:

```
gcc -c -fdump-ada-spec -C ./test.h
```

The result is an Ada spec file called `test_h.ads`:

Listing 18: `test_h.ads`

```

1 pragma Ada_2005;
2 pragma Style_Checks (Off);
3
4 with Interfaces.C; use Interfaces.C;
5
6 package test_h is
7
8   func_cnt : aliased int; -- ./test.h:3
9   pragma Import (C, func_cnt, "func_cnt");
10
11  function my_func (arg1 : int) return int; -- ./test.h:5
12  pragma Import (C, my_func, "my_func");
13
14 end test_h;

```

Now we simply refer to this `test_h` package in our Ada application:

Listing 19: `show_c_func.adb`

```

1 with Interfaces.C; use Interfaces.C;
2 with Ada.Text_IO; use Ada.Text_IO;
3 with test_h;       use test_h;
4
5 procedure Show_C_Func is
6   V : int;
7 begin
8   V := my_func (1);
9   V := my_func (2);

```

(continues on next page)

(continued from previous page)

```
10  V := my_func (3);
11  Put_Line ("Result is " & int'Image (V));
12
13  Put_Line ("Function was called "
14      & int'Image (func_cnt)
15      & " times");
16 end Show_C_Func;
```

You can specify the name of the parent unit for the bindings you're creating as the operand to `fdump-ada-spec`:

```
gcc -c -fdump-ada-spec -fada-spec-parent=Ext_C_Code -C ./test.h
```

This creates the file `ext_c_code-test_h.ads`:

Listing 20: `ext_c_code-test_h.ads`

```
1 package Ext_C_Code.test_h is
2
3     -- automatic generated bindings...
4
5 end Ext_C_Code.test_h;
```

17.5.1 Adapting bindings

The compiler does the best it can when creating bindings for a C header file. However, sometimes it has to guess about the translation and the generated bindings don't always match our expectations. For example, this can happen when creating bindings for functions that have pointers as arguments. In this case, the compiler may use `System.Address` as the type of one or more pointers. Although this approach works fine (as we'll see later), this is usually not how a human would interpret the C header file. The following example illustrates this issue.

Let's start with this C header file:

Listing 21: `test.h`

```
1 struct test;
2
3 struct test * test_create(void);
4
5 void test_destroy(struct test *t);
6
7 void test_reset(struct test *t);
8
9 void test_set_name(struct test *t, char *name);
10
11 void test_set_address(struct test *t, char *address);
12
13 void test_display(const struct test *t);
```

And the corresponding C implementation:

Listing 22: `test.c`

```
1 #include <stdlib.h>
2 #include <string.h>
3 #include <stdio.h>
```

(continues on next page)

(continued from previous page)

```

5 #include "test.h"
6
7 struct test {
8     char name[80];
9     char address[120];
10 };
11
12 static size_t
13 strlcpy(char *dst, const char *src, size_t dstsize)
14 {
15     size_t len = strlen(src);
16     if (dstsize) {
17         size_t bl = (len < dstsize-1 ? len : dstsize-1);
18         ((char*)memcpy(dst, src, bl))[bl] = 0;
19     }
20     return len;
21 }
22
23 struct test * test_create(void)
24 {
25     return malloc (sizeof (struct test));
26 }
27
28 void test_destroy(struct test *t)
29 {
30     if (t != NULL) {
31         free(t);
32     }
33 }
34
35 void test_reset(struct test *t)
36 {
37     t->name[0] = '\0';
38     t->address[0] = '\0';
39 }
40
41 void test_set_name(struct test *t, char *name)
42 {
43     strlcpy(t->name, name, sizeof(t->name));
44 }
45
46 void test_set_address(struct test *t, char *address)
47 {
48     strlcpy(t->address, address, sizeof(t->address));
49 }
50
51 void test_display(const struct test *t)
52 {
53     printf("Name: %s\n", t->name);
54     printf("Address: %s\n", t->address);
55 }
```

Next, we'll create our bindings:

```
gcc -c -fdump-ada-spec -C ./test.h
```

This creates the following specification in test_h.ads:

Listing 23: test_h.ads

```

1  pragma Ada_2005;
2  pragma Style_Checks (Off);
3
4  with Interfaces.C; use Interfaces.C;
5  with System;
6  with Interfaces.C.Strings;
7
8  package test_h is
9
10   -- skipped empty struct test
11
12   function test_create return System.Address; -- ./test.h:5
13   pragma Import (C, test_create, "test_create");
14
15   procedure test_destroy (arg1 : System.Address); -- ./test.h:7
16   pragma Import (C, test_destroy, "test_destroy");
17
18   procedure test_reset (arg1 : System.Address); -- ./test.h:9
19   pragma Import (C, test_reset, "test_reset");
20
21   procedure test_set_name (arg1 : System.Address; arg2 : Interfaces.C.Strings.
22   chars_ptr); -- ./test.h:11
23   pragma Import (C, test_set_name, "test_set_name");
24
25   procedure test_set_address (arg1 : System.Address; arg2 : Interfaces.C.Strings.
26   chars_ptr); -- ./test.h:13
27   pragma Import (C, test_set_address, "test_set_address");
28
29   procedure test_display (arg1 : System.Address); -- ./test.h:15
30   pragma Import (C, test_display, "test_display");
31
32 end test_h;

```

As we can see, the binding generator completely ignores the declaration `struct test` and all references to the `test` struct are replaced by addresses (`System.Address`). Nevertheless, these bindings are good enough to allow us to create a test application in Ada:

Listing 24: show_automatic_c_struct_bindings.adb

```

1  with Interfaces.C;           use Interfaces.C;
2  with Interfaces.C.Strings;   use Interfaces.C.Strings;
3  with Ada.Text_IO;           use Ada.Text_IO;
4  with test_h;                use test_h;
5
6  with System;
7
8  procedure Show_Automatic_C_Struct_Bindings is
9
10   Name    : constant chars_ptr := 
11     New_String ("John Doe");
12   Address : constant chars_ptr := 
13     New_String ("Small Town");
14
15   T : System.Address := test_create;
16
17 begin
18   test_reset (T);
19   test_set_name (T, Name);
20   test_set_address (T, Address);

```

(continues on next page)

(continued from previous page)

```

21      test_display (T);
22      test_destroy (T);
23
24 end Show_Automatic_C_Struct_Bindings;

```

We can successfully bind our C code with Ada using the automatically-generated bindings, but they aren't ideal. Instead, we would prefer Ada bindings that match our (human) interpretation of the C header file. This requires manual analysis of the header file. The good news is that we can use the automatic generated bindings as a starting point and adapt them to our needs. For example, we can:

1. Define a Test type based on `System.Address` and use it in all relevant functions.
2. Remove the `test_` prefix in all operations on the Test type.

This is the resulting specification:

Listing 25: adapted_test_h.ads

```

1  with Interfaces.C; use Interfaces.C;
2  with System;
3  with Interfaces.C.Strings;
4
5 package adapted_test_h is
6
7   type Test is new System.Address;
8
9   function Create return Test;
10  pragma Import (C, Create, "test_create");
11
12  procedure Destroy (T : Test);
13  pragma Import (C, Destroy, "test_destroy");
14
15  procedure Reset (T : Test);
16  pragma Import (C, Reset, "test_reset");
17
18  procedure Set_Name (T      : Test;
19                      Name   : Interfaces.C.Strings.chars_ptr); -- ./test.h:11
20  pragma Import (C, Set_Name, "test_set_name");
21
22  procedure Set_Address (T       : Test;
23                        Address : Interfaces.C.Strings.chars_ptr);
24  pragma Import (C, Set_Address, "test_set_address");
25
26  procedure Display (T : Test); -- ./test.h:15
27  pragma Import (C, Display, "test_display");
28
29 end adapted_test_h;

```

And this is the corresponding Ada body:

Listing 26: show_adapted_c_struct_bindings.adb

```

1  with Interfaces.C;           use Interfaces.C;
2  with Interfaces.C.Strings;   use Interfaces.C.Strings;
3  with adapted_test_h;        use adapted_test_h;
4
5  with System;
6
7  procedure Show_Adapted_C_Struct_Bindings is
8
9    Name      : constant chars_ptr := 

```

(continues on next page)

(continued from previous page)

```
10  New_String ("John Doe");
11  Address : constant chars_ptr :=
12      New_String ("Small Town");
13
14  T : Test := Create;
15
16 begin
17  Reset (T);
18  Set_Name (T, Name);
19  Set_Address (T, Address);
20
21  Display (T);
22  Destroy (T);
23 end Show_Adapted_C_Struct_Bindings;
```

Now we can use the `Test` type and its operations in a clean, readable way.

CHAPTER
EIGHTEEN

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a large and ill-defined concept in programming languages and one that tends to encompass many different meanings because different languages often implement their own vision of it, with similarities and differences from the implementations in other languages.

However, one model mostly "won" the battle of what object-oriented means, if only by sheer popularity. It's the model used in the Java programming language, which is very similar to the one used by C++. Here are some defining characteristics:

- Type derivation and extension: Most object oriented languages allow the user to add fields to derived types.
- Subtyping: Objects of a type derived from a base type can, in some instances, be substituted for objects of the base type.
- Runtime polymorphism: Calling a subprogram, usually called a *method*, attached to an object type can dispatch at runtime depending on the exact type of the object.
- Encapsulation: Objects can hide some of their data.
- Extensibility: People from the "outside" of your package, or even your whole library, can derive from your object types and define their own behaviors.

Ada dates from before object-oriented programming was as popular as it is today. Some of the mechanisms and concepts from the above list were in the earliest version of Ada even before what we would call OOP was added:

- As we saw, encapsulation is not implemented at the type level in Ada, but instead at the package level.
- Subtyping can be implemented using, well, subtypes, which have a full and permissive static substitutability model. The substitution will fail at runtime if the dynamic constraints of the subtype are not fulfilled.
- Runtime polymorphism can be implemented using variant records.

However, this lists leaves out type extensions, if you don't consider variant records, and extensibility.

The 1995 revision of Ada added a feature filling the gaps, which allowed people to program following the object-oriented paradigm in an easier fashion. This feature is called *tagged types*.

Note: It's possible to program in Ada without ever creating tagged types. If that's your preferred style of programming or you have no specific use for tagged types, feel free to not use them, as is the case for many features of Ada.

However, they can be the best way to express solutions to certain problems and they may be the best way to solve your problem. If that's the case, read on!

18.1 Derived types

Before presenting tagged types, we should discuss a topic we have brushed on, but not really covered, up to now:

You can create one or more new types from every type in Ada. Type derivation is built into the language.

Listing 1: newtypes.ads

```

1 package Newtypes is
2     type Point is record
3         X, Y : Integer;
4     end record;
5
6     type New_Point is new Point;
7 end Newtypes;
```

Type derivation is useful to enforce strong typing because the type system treats the two types as incompatible.

But the benefits are not limited to that: you can inherit things from the type you derive from. You not only inherit the representation of the data, but you can also inherit behavior.

When you inherit a type you also inherit what are called *primitive operations*. A primitive operation (or just a *primitive*) is a subprogram attached to a type. Ada defines primitives as subprograms defined in the same scope as the type.

Attention: A subprogram will only become a primitive of the type if:

1. The subprogram is declared in the same scope as the type and
2. The type and the subprogram are declared in a package

Listing 2: primitives.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Primitives is
4     package Week is
5         type Days is (Monday, Tuesday, Wednesday,
6                         Thursday, Friday,
7                         Saturday, Sunday);
8
9         -- Print_Day is a primitive
10        -- of the type Days
11         procedure Print_Day (D : Days);
12     end Week;
13
14     package body Week is
15         procedure Print_Day (D : Days) is
16             begin
17                 Put_Line (Days'Image (D));
18             end Print_Day;
19     end Week;
20
21     use Week;
22     type Weekend_Days is new
23         Days range Saturday .. Sunday;
```

(continues on next page)

(continued from previous page)

```

25   -- A procedure Print_Day is automatically
26   -- inherited here. It is as if the procedure
27   --
28   -- procedure Print_Day (D : Weekend_Days);
29   --
30   -- has been declared with the same body
31
32   Sat : Weekend_Days := Saturday;
33 begin
34   Print_Day (Sat);
35 end Primitives;

```

Build output

```

primitives.adb:11:15: warning: procedure "Print_Day" is not referenced [-gnatwu]
primitives.adb:32:03: warning: "Sat" is not modified, could be declared constant [-gnatwk]

```

Runtime output

```
SATURDAY
```

This kind of inheritance can be very useful, and is not limited to record types (you can use it on discrete types, as in the example above), but it's only superficially similar to object-oriented inheritance:

- Records can't be extended using this mechanism alone. You also can't specify a new representation for the new type: it will **always** have the same representation as the base type.
- There's no facility for dynamic dispatch or polymorphism. Objects are of a fixed, static type.

There are other differences, but it's not useful to list them all here. Just remember that this is a kind of inheritance you can use if you only want to statically inherit behavior without duplicating code or using composition, but a kind you can't use if you want any dynamic features that are usually associated with OOP.

18.2 Tagged types

The 1995 revision of the Ada language introduced tagged types to fulfil the need for an unified solution that allows programming in an object-oriented style similar to the one described at the beginning of this chapter.

Tagged types are very similar to normal records except that some functionality is added:

- Types have a *tag*, stored inside each object, that identifies the *runtime type*¹⁹ of that object.
- Primitives can dispatch. A primitive on a tagged type is what you would call a *method* in Java or C++. If you derive a base type and override a primitive of it, you can often call it on an object with the result that which primitive is called depends on the exact runtime type of the object.
- Subtyping rules are introduced allowing a tagged type derived from a base type to be statically compatible with the base type.

Let's see our first tagged type declarations:

¹⁹ https://en.wikipedia.org/wiki/Run-time_type_information

Listing 3: p.ads

```

1 package P is
2     type My_Class is tagged null record;
3         -- Just like a regular record, but
4         -- with tagged qualifier
5
6         -- Methods are outside of the type
7         -- definition:
8
9     procedure Foo (Self : in out My_Class);
10        -- If you define a procedure taking a
11        -- My_Class argument in the same package,
12        -- it will be a method.
13
14        -- Here's how you derive a tagged type:
15
16    type Derived is new My_Class with record
17        A : Integer;
18            -- You can add fields in derived types.
19    end record;
20
21    overriding procedure Foo (Self : in out Derived);
22        -- The "overriding" qualifier is optional,
23        -- but if it is present, it must be valid.
24 end P;

```

Listing 4: p.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4     procedure Foo (Self : in out My_Class) is
5 begin
6     Put_Line ("In My_Class.Foo");
7 end Foo;
8
9     procedure Foo (Self : in out Derived) is
10 begin
11     Put_Line ("In Derived.Foo, A = "
12             & Integer'Image (Self.A));
13 end Foo;
14 end P;

```

18.3 Classwide types

To remain consistent with the rest of the language, a new notation needed to be introduced to say "This object is of this type or any descendent derives tagged type".

In Ada, we call this the *classwide type*. It's used in OOP as soon as you need polymorphism. For example, you can't do the following:

Listing 5: main.adb

```

1 with P; use P;
2
3 procedure Main is
4

```

(continues on next page)

(continued from previous page)

```

5   01 : My_Class;
6   -- Declaring an object of type My_Class
7
8   02 : Derived := (A => 12);
9   -- Declaring an object of type Derived
10
11  03 : My_Class := 02;
12  -- INVALID: Trying to assign a value
13  -- of type derived to a variable of
14  -- type My_Class.
15 begin
16   null;
17 end Main;

```

Build output

```

main.adb:11:21: error: expected type "My_Class" defined at p.ads:2
main.adb:11:21: error: found type "Derived" defined at p.ads:16
gprbuild: *** compilation phase failed

```

This is because an object of a type T is exactly of the type T, whether T is tagged or not. What you want to say as a programmer is "I want O3 to be able to hold an object of type My_Class or any type descending from My_Class". Here's how you do that:

Listing 6: main.adb

```

1 with P; use P;
2
3 procedure Main is
4   01 : My_Class;
5   -- Declare an object of type My_Class
6
7   02 : Derived := (A => 12);
8   -- Declare an object of type Derived
9
10  03 : My_Class'Class := 02;
11  -- Now valid: My_Class'Class designates
12  -- the classwide type for My_Class,
13  -- which is the set of all types
14  -- descending from My_Class (including
15  -- My_Class).
16 begin
17   null;
18 end Main;

```

Build output

```

main.adb:4:04: warning: variable "01" is not referenced [-gnatwu]
main.adb:7:04: warning: "02" is not modified, could be declared constant [-gnatwk]
main.adb:10:04: warning: variable "03" is not referenced [-gnatwu]

```

Attention: Because an object of a classwide type can be the size of any descendent of its base type, it has an unknown size. It's therefore an indefinite type, with the expected restrictions:

- It can't be stored as a field/component of a record
- An object of a classwide type needs to be initialized immediately (you can't specify the constraints of such a type in any way other than by initializing it).

18.4 Dispatching operations

We saw that you can override operations in types derived from another tagged type. The eventual goal of OOP is to make a dispatching call: a call to a primitive (method) that depends on the exact type of the object.

But, if you think carefully about it, a variable of type `My_Class` always contains an object of exactly that type. If you want to have a variable that can contain a `My_Class` or any derived type, it has to be of type `My_Class'Class`.

In other words, to make a dispatching call, you must first have an object that can be either of a type or any type derived from this type, namely an object of a classwide type.

Listing 7: main.adb

```

1  with P; use P;
2
3  procedure Main is
4      01 : My_Class;
5      -- Declare an object of type My_Class
6
7      02 : Derived := (A => 12);
8      -- Declare an object of type Derived
9
10     03 : My_Class'Class := 02;
11
12     04 : My_Class'Class := 01;
13 begin
14     Foo (01);
15     -- Non dispatching: Calls My_Class.Foo
16     Foo (02);
17     -- Non dispatching: Calls Derived.Foo
18     Foo (03);
19     -- Dispatching: Calls Derived.Foo
20     Foo (04);
21     -- Dispatching: Calls My_Class.Foo
22 end Main;

```

Runtime output

```

In My_Class.Foo
In Derived.Foo, A =  12
In Derived.Foo, A =  12
In My_Class.Foo

```

Attention

You can convert an object of type `Derived` to an object of type `My_Class`. This is called a *view conversion* in Ada parlance and is useful, for example, if you want to call a parent method.

In that case, the object really is converted to a `My_Class` object, which means its tag is changed. Since tagged objects are always passed by reference, you can use this kind of conversion to modify the state of an object: changes to converted object will affect the original one.

Listing 8: main.adb

```

1  with P; use P;
2
3  procedure Main is

```

(continues on next page)

(continued from previous page)

```

4 01 : Derived := (A => 12);
5   -- Declare an object of type Derived
6
7 02 : My_Class := My_Class (01);
8
9 03 : My_Class'Class := 02;
10 begin
11   Foo (01);
12   -- Non dispatching: Calls Derived.Foo
13   Foo (02);
14   -- Non dispatching: Calls My_Class.Foo
15
16   Foo (03);
17   -- Dispatching: Calls My_Class.Foo
18 end Main;

```

Runtime output

```

In Derived.Foo, A = 12
In My_Class.Foo
In My_Class.Foo

```

18.5 Dot notation

You can also call primitives of tagged types with a notation that's more familiar to object oriented programmers. Given the `Foo` primitive above, you can also write the above program this way:

Listing 9: main.adb

```

1 with P; use P;
2
3 procedure Main is
4   01 : My_Class;
5   -- Declare an object of type My_Class
6
7   02 : Derived := (A => 12);
8   -- Declare an object of type Derived
9
10  03 : My_Class'Class := 02;
11
12  04 : My_Class'Class := 01;
13 begin
14   01.Foo;
15   -- Non dispatching: Calls My_Class.Foo
16   02.Foo;
17   -- Non dispatching: Calls Derived.Foo
18   03.Foo;
19   -- Dispatching: Calls Derived.Foo
20   04.Foo;
21   -- Dispatching: Calls My_Class.Foo
22 end Main;

```

Runtime output

```

In My_Class.Foo
In Derived.Foo, A = 12

```

(continues on next page)

(continued from previous page)

```
In Derived.Foo, A = 12
In My_Class.Foo
```

If the dispatching parameter of a primitive is the first parameter, which is the case in our examples, you can call the primitive using the dot notation. Any remaining parameter are passed normally:

Listing 10: main.adb

```

1  with P; use P;
2
3  procedure Main is
4    package Extend is
5      type D2 is new Derived with null record;
6
7      procedure Bar (Self : in out D2;
8                      Val :           Integer);
9    end Extend;
10
11  package body Extend is
12    procedure Bar (Self : in out D2;
13                  Val :           Integer) is
14      begin
15        Self.A := Self.A + Val;
16      end Bar;
17    end Extend;
18
19    use Extend;
20
21    Obj : D2 := (A => 15);
22  begin
23    Obj.Bar (2);
24    Obj.Foo;
25  end Main;
```

Runtime output

```
In Derived.Foo, A = 17
```

18.6 Private & Limited

We've seen previously (in the *Privacy* (page 107) chapter) that types can be declared limited or private. These encapsulation techniques can also be applied to tagged types, as we'll see in this section.

This is an example of a tagged private type:

Listing 11: p.ads

```

1  package P is
2    type T is tagged private;
3  private
4    type T is tagged record
5      E : Integer;
6    end record;
7  end P;
```

This is an example of a tagged limited type:

Listing 12: p.ads

```

1 package P is
2   type T is tagged limited record
3     E : Integer;
4   end record;
5 end P;

```

Naturally, you can combine both *limited* and *private* types and declare a tagged limited private type:

Listing 13: p.ads

```

1 package P is
2   type T is tagged limited private;
3
4   procedure Init (A : in out T);
5 private
6   type T is tagged limited record
7     E : Integer;
8   end record;
9 end P;

```

Listing 14: p.adb

```

1 package body P is
2
3   procedure Init (A : in out T) is
4   begin
5     A.E := 0;
6   end Init;
7
8 end P;

```

Listing 15: main.adb

```

1 with P; use P;
2
3 procedure Main is
4   T1, T2 : T;
5 begin
6   T1.Init;
7   T2.Init;
8
9   -- The following line doesn't work
10  -- because type T is private:
11  --
12  -- T1.E := 0;
13
14  -- The following line doesn't work
15  -- because type T is limited:
16  --
17  -- T2 := T1;
18 end Main;

```

Note that the code in the Main procedure above presents two assignments that trigger compilation errors because type T is limited private. In fact, you cannot:

- assign to T1.E directly because type T is private;
- assign T1 to T2 because type T is limited.

In this case, there's no distinction between tagged and non-tagged types: these compilation errors would also occur for non-tagged types.

18.7 Classwide access types

In this section, we'll discuss an useful pattern for object-oriented programming in Ada: class-wide access type. Let's start with an example where we declare a tagged type T and a derived type T_New:

Listing 16: p.ads

```

1 package P is
2     type T is tagged null record;
3
4     procedure Show (Dummy : T);
5
6     type T_New is new T with null record;
7
8     procedure Show (Dummy : T_New);
9 end P;
```

Listing 17: p.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5     procedure Show (Dummy : T) is
6     begin
7         Put_Line ("Using type "
8                   & T'External_Tag);
9     end Show;
10
11    procedure Show (Dummy : T_New) is
12    begin
13        Put_Line ("Using type "
14                  & T_New'External_Tag);
15    end Show;
16
17 end P;
```

Note that we're using null records for both types T and T_New. Although these types don't actually have any component, we can still use them to demonstrate dispatching. Also note that the example above makes use of the '`External_Tag`' attribute in the implementation of the `Show` procedure to get a string for the corresponding tagged type.

As we've seen before, we must use a classwide type to create objects that can make dispatching calls. In other words, objects of type `T'Class` will dispatch. For example:

Listing 18: dispatching_example.adb

```

1 with P; use P;
2
3 procedure Dispatching_Example is
4     T2          :          T_New;
5     T_Dispatch : constant T'Class := T2;
6 begin
7     T_Dispatch.Show;
8 end Dispatching_Example;
```

Runtime output

Using type P.T_NEW

A more useful application is to declare an array of objects that can dispatch. For example, we'd like to declare an array T_Arr, loop over this array and dispatch according to the actual type of each individual element:

```
for I in T_Arr'Range loop
    T_Arr (I).Show;
    -- Call Show procedure according
    -- to actual type of T_Arr (I)
end loop;
```

However, it's not possible to declare an array of type T'Class directly:

Listing 19: classwide_compilation_error.adb

```
1 with P; use P;
2
3 procedure Classwide_Compilation_Error is
4     T_Arr : array (1 .. 2) of T'Class;
5     ^
6     -- Compilation Error!
7 begin
8     for I in T_Arr'Range loop
9         T_Arr (I).Show;
10    end loop;
11 end Classwide_Compilation_Error;
```

Build output

```
classwide_compilation_error.adb:4:31: error: unconstrained element type in array
^declaration
gprbuild: *** compilation phase failed
```

In fact, it's impossible for the compiler to know which type would actually be used for each element of the array. However, if we use dynamic allocation via access types, we can allocate objects of different types for the individual elements of an array T_Arr. We do this by using classwide access types, which have the following format:

```
type T_Class is access T'Class;
```

We can rewrite the previous example using the T_Class type. In this case, dynamically allocated objects of this type will dispatch according to the actual type used during the allocation. Also, let's introduce an Init procedure that won't be overridden for the derived T_New type. This is the adapted code:

Listing 20: p.ads

```
1 package P is
2     type T is tagged record
3         E : Integer;
4     end record;
5
6     type T_Class is access T'Class;
7
8     procedure Init (A : in out T);
9
10    procedure Show (Dummy : T);
11
12    type T_New is new T with null record;
```

(continues on next page)

(continued from previous page)

```
13  procedure Show (Dummy : T_New);
14
15 end P;
```

Listing 21: p.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3 package body P is
4
5   procedure Init (A : in out T) is
6   begin
7     Put_Line ("Initializing type T...");
8     A.E := 0;
9   end Init;
10
11  procedure Show (Dummy : T) is
12  begin
13    Put_Line ("Using type "
14      & T'External_Tag);
15  end Show;
16
17  procedure Show (Dummy : T_New) is
18  begin
19    Put_Line ("Using type "
20      & T_New'External_Tag);
21  end Show;
22
23 end P;
```

Listing 22: main.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with P;           use P;
3
4  procedure Main is
5    T_Arr : array (1 .. 2) of T_Class;
6  begin
7    T_Arr (1) := new T;
8    T_Arr (2) := new T_New;
9
10   for I in T_Arr'Range loop
11     Put_Line ("Element # "
12       & Integer'Image (I));
13
14     T_Arr (I).Init;
15     T_Arr (I).Show;
16
17     Put_Line ("-----");
18   end loop;
19 end Main;
```

Runtime output

```
Element # 1
Initializing type T...
Using type P.T
-----
Element # 2
```

(continues on next page)

(continued from previous page)

```
Initializing type T...
Using type P.T_NEW
-----
```

In this example, the first element (`T_Arr (1)`) is of type `T`, while the second element is of type `T_New`. When running the example, the `Init` procedure of type `T` is called for both elements of the `T_Arr` array, while the call to the `Show` procedure selects the corresponding procedure according to the type of each element of `T_Arr`.

STANDARD LIBRARY: CONTAINERS

In previous chapters, we've used arrays as the standard way to group multiple objects of a specific data type. In many cases, arrays are good enough for manipulating those objects. However, there are situations that require more flexibility and more advanced operations. For those cases, Ada provides support for containers — such as vectors and sets — in its standard library.

We present an introduction to containers here. For a list of all containers available in Ada, see [Appendix B](#) (page 257).

19.1 Vectors

In the following sections, we present a general overview of vectors, including instantiation, initialization, and operations on vector elements and vectors.

19.1.1 Instantiation

Here's an example showing the instantiation and declaration of a vector V:

Listing 1: show_vector_inst.adb

```
1 with Ada.Containers.Vectors;
2
3 procedure Show_Vector_Inst is
4
5   package Integer_Vectors is new
6     Ada.Containers.Vectors
7       (Index_Type    => Natural,
8        Element_Type => Integer);
9
10  V : Integer_Vectors.Vector;
11 begin
12   null;
13 end Show_Vector_Inst;
```

Build output

```
show_vector_inst.adb:10:04: warning: variable "V" is not referenced [-gnatwu]
```

Containers are based on generic packages, so we can't simply declare a vector as we would declare an array of a specific type:

```
A : array (1 .. 10) of Integer;
```

Instead, we first need to instantiate one of those packages. We **with** the container package (`Ada.Containers.Vectors` in this case) and instantiate it to create an instance of the generic package for the desired type. Only then can we declare the vector using the type from the instantiated package. This instantiation needs to be done for any container type from the standard library.

In the instantiation of `Integer_Vectors`, we indicate that the vector contains elements of **Integer** type by specifying it as the `Element_Type`. By setting `Index_Type` to **Natural**, we specify that the allowed range includes all natural numbers. We could have used a more restrictive range if desired.

19.1.2 Initialization

One way to initialize a vector is from a concatenation of elements. We use the `&` operator, as shown in the following example:

Listing 2: `show_vector_init.adb`

```
1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Init is
7
8    package Integer_Vectors is new
9      Ada.Containers.Vectors
10     (Index_Type    => Natural,
11      Element_Type => Integer);
12
13  use Integer_Vectors;
14
15  V : Vector := 20 & 10 & 0 & 13;
16 begin
17   Put_Line ("Vector has "
18             & Count_Type'Image (V.Length)
19             & " elements");
20 end Show_Vector_Init;
```

Build output

```
show_vector_init.adb:15:04: warning: "V" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Vector has 4 elements
```

We specify **use** `Integer_Vectors`, so we have direct access to the types and operations from the instantiated package. Also, the example introduces another operation on the vector: `Length`, which retrieves the number of elements in the vector. We can use the dot notation because `Vector` is a tagged type, allowing us to write either `V.Length` or `Length (V)`.

19.1.3 Appending and prepending elements

You add elements to a vector using the Prepend and Append operations. As the names suggest, these operations add elements to the beginning or end of a vector, respectively. For example:

Listing 3: show_vector_append.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Append is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10         (Index_Type  => Natural,
11          Element_Type => Integer);
12
13     use Integer_Vectors;
14
15     V : Vector;
16
17 begin
18     Put_Line ("Appending some elements to the vector...");
19     V.Append (20);
20     V.Append (10);
21     V.Append (0);
22     V.Append (13);
23     Put_Line ("Finished appending.");
24
25     Put_Line ("Prepending some elements to the vector...");
26     V.Prepend (30);
27     V.Prepend (40);
28     V.Prepend (100);
29     Put_Line ("Finished prepending.");
30
31     Put_Line ("Vector has "
32             & Count_Type'Image (V.Length)
33             & " elements");
34 end Show_Vector_Append;

```

Runtime output

```

Appending some elements to the vector...
Finished appending.
Prepending some elements to the vector...
Finished prepending.
Vector has 7 elements

```

This example puts elements into the vector in the following sequence: (100, 40, 30, 20, 10, 0, 13).

The Reference Manual specifies that the worst-case complexity must be:

- O(log N) for the Append operation, and
- O(N log N) for the Prepend operation.

19.1.4 Accessing first and last elements

We access the first and last elements of a vector using the `First_Element` and `Last_Element` functions. For example:

Listing 4: `show_vector_first_last_element.adb`

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_First_Last_Element is
7
8    package Integer_Vectors is new
9      Ada.Containers.Vectors
10     (Index_Type  => Natural,
11      Element_Type => Integer);
12
13  use Integer_Vectors;
14
15  function Img (I : Integer)      return String
16    renames Integer'Image;
17  function Img (I : Count_Type) return String
18    renames Count_Type'Image;
19
20  V : Vector := 20 & 10 & 0 & 13;
21 begin
22   Put_Line ("Vector has "
23             & Img (V.Length)
24             & " elements");
25
26   -- Using V.First_Element to
27   -- retrieve first element
28   Put_Line ("First element is "
29             & Img (V.First_Element));
30
31   -- Using V.Last_Element to
32   -- retrieve last element
33   Put_Line ("Last element is "
34             & Img (V.Last_Element));
35 end Show_Vector_First_Last_Element;

```

Build output

```
show_vector_first_last_element.adb:20:04: warning: "V" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Vector has 4 elements
First element is 20
Last element is 13
```

You can swap elements by calling the procedure `Swap` and retrieving a reference (a *cursor*) to the first and last elements of the vector by calling `First` and `Last`. A cursor allows us to iterate over a container and process individual elements from it.

With these operations, we're able to write code to swap the first and last elements of a vector:

Listing 5: show_vector_first_last_element.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_First_Last_Element is
7
8    package Integer_Vectors is new
9      Ada.Containers.Vectors
10     (Index_Type    => Natural,
11      Element_Type => Integer);
12
13  use Integer_Vectors;
14
15  function Img (I : Integer) return String
16    renames Integer'Image;
17
18  V : Vector := 20 & 10 & 0 & 13;
19 begin
20   -- We use V.First and V.Last to retrieve
21   -- cursor for first and last elements.
22   -- We use V.Swap to swap elements.
23  V.Swap (V.First, V.Last);
24
25  Put_Line ("First element is now "
26            & Img (V.First_Element));
27  Put_Line ("Last element is now "
28            & Img (V.Last_Element));
29 end Show_Vector_First_Last_Element;

```

Runtime output

```

First element is now 13
Last element is now 20

```

19.1.5 Iterating

The easiest way to iterate over a container is to use a **for E of Our_Container** loop. This gives us a reference (E) to the element at the current position. We can then use E directly. For example:

Listing 6: show_vector_iteration.adb

```

1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Vector_Iteration is
6
7    package Integer_Vectors is new
8      Ada.Containers.Vectors
9      (Index_Type    => Natural,
10       Element_Type => Integer);
11
12  use Integer_Vectors;
13
14  function Img (I : Integer) return String

```

(continues on next page)

(continued from previous page)

```

15 renames Integer'Image;
16
17 V : Vector := 20 & 10 & 0 & 13;
18 begin
19   Put_Line ("Vector elements are: ");
20
21   -- Using for ... of loop to iterate:
22   --
23   for E of V loop
24     Put_Line ("- " & Img (E));
25   end loop;
26
27 end Show_Vector_Iteration;

```

Build output

```
show_vector_iteration.adb:17:04: warning: "V" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Vector elements are:
- 20
- 10
- 0
- 13
```

This code displays each element from the vector V.

Because we're given a reference, we can display not only the value of an element but also modify it. For example, we could easily write a loop to add one to each element of vector V:

```

for E of V loop
  E := E + 1;
end loop;
```

We can also use indices to access vector elements. The format is similar to a loop over array elements: we use a **for I in <range>** loop. The range is provided by V.First_Index and V.Last_Index. We can access the current element by using it as an array index: V(I). For example:

Listing 7: show_vector_index_iteration.adb

```

1 with Ada.Containers.Vectors;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Vector_Index_Iteration is
6
7   package Integer_Vectors is new
8     Ada.Containers.Vectors
9       (Index_Type    => Natural,
10        Element_Type => Integer);
11
12   use Integer_Vectors;
13
14   V : Vector := 20 & 10 & 0 & 13;
15 begin
16   Put_Line ("Vector elements are: ");

```

(continues on next page)

(continued from previous page)

```

17
18      -- Using indices in a "for I in ..." loop
19      -- to iterate:
20
21
22  for I in V.First_Index .. V.Last_Index loop
23      -- Displaying current index I
24      Put (" - ["
25          & Extended_Index'Image (I)
26          & "] ");
27
28      Put (Integer'Image (V (I)));
29
30      -- We could also use the V.Element (I)
31      -- function to retrieve the element at
32      -- the current index I
33
34      New_Line;
35  end loop;
36
37 end Show_Vector_Index_Iteration;

```

Build output

```
show_vector_index_iteration.adb:14:04: warning: "V" is not modified, could be undeclared constant [-gnatwk]
```

Runtime output

```
Vector elements are:
- [ 0] 20
- [ 1] 10
- [ 2] 0
- [ 3] 13
```

Here, in addition to displaying the vector elements, we're also displaying each index, I, just like what we can do for array indices. Also, we can access the element by using either the short form `V (I)` or the longer form `V.Element (I)` but not `V.I`.

As mentioned in the previous section, you can use cursors to iterate over containers. For this, use the function `Iterate`, which retrieves a cursor for each position in the vector. The corresponding loop has the format `for C in V.Iterate loop`. Like the previous example using indices, you can again access the current element by using the cursor as an array index: `V (C)`. For example:

Listing 8: show_vector_cursor_iteration.adb

```

1 with Ada.Containers.Vectors;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Vector_Cursor_Iteration is
6
7     package Integer_Vectors is new
8         Ada.Containers.Vectors
9             (Index_Type    => Natural,
10              Element_Type => Integer);
11
12     use Integer_Vectors;
13
14     V : Vector := 20 & 10 & 0 & 13;

```

(continues on next page)

(continued from previous page)

```

15 begin
16   Put_Line ("Vector elements are: ");
17
18   --
19   -- Use a cursor to iterate in a loop:
20   --
21   for C in V.Iterate loop
22     -- Using To_Index function to retrieve
23     -- the index for the cursor position
24     Put ("- ["
25       & Extended_Index'Image (To_Index (C))
26       & "] ");
27
28     Put (Integer'Image (V (C)));
29
30     -- We could use Element (C) to retrieve
31     -- the vector element for the cursor
32     -- position
33
34     New_Line;
35   end loop;
36
37   -- Alternatively, we could iterate with a
38   -- while-loop:
39   --
40   -- declare
41   --   C : Cursor := V.First;
42   -- begin
43   --   while C /= No_Element loop
44   --     some processing here...
45   --
46   --     C := Next (C);
47   --   end loop;
48   -- end;
49
50 end Show_Vector_Cursor_Iteration;

```

Build output

```
show_vector_cursor_iteration.adb:14:04: warning: "V" is not modified, could be
→ declared constant [-gnatwk]
```

Runtime output

```
Vector elements are:
- [ 0] 20
- [ 1] 10
- [ 2] 0
- [ 3] 13
```

Instead of accessing an element in the loop using `V (C)`, we could also have used the longer form `Element (C)`. In this example, we're using the function `To_Index` to retrieve the index corresponding to the current cursor.

As shown in the comments after the loop, we could also use a `while ... loop` to iterate over the vector. In this case, we would start with a cursor for the first element (retrieved by calling `V.First`) and then call `Next (C)` to retrieve a cursor for subsequent elements. `Next (C)` returns `No_Element` when the cursor reaches the end of the vector.

You can directly modify the elements using a reference. This is what it looks like when using both indices and cursors:

```
-- Modify vector elements using index
for I in V.First_Index .. V.Last_Index loop
    V (I) := V (I) + 1;
end loop;

-- Modify vector elements using cursor
for C in V.Iterate loop
    V (C) := V (C) + 1;
end loop;
```

The Reference Manual requires that the worst-case complexity for accessing an element be $O(\log N)$.

Another way of modifying elements of a vector is using a *process procedure*, which takes an individual element and does some processing on it. You can call `Update_Element` and pass both a cursor and an access to the process procedure. For example:

Listing 9: show_vector_update.adb

```
1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Vector_Update is
6
7      package Integer_Vectors is new
8          Ada.Containers.Vectors
9              (Index_Type    => Natural,
10               Element_Type => Integer);
11
12     use Integer_Vectors;
13
14     procedure Add_One (I : in out Integer) is
15     begin
16         I := I + 1;
17     end Add_One;
18
19     V : Vector := 20 & 10 & 12;
20 begin
21     --
22     -- Use V.Update_Element to process elements
23     --
24     for C in V.Iterate loop
25         V.Update_Element (C, Add_One'Access);
26     end loop;
27
28 end Show_Vector_Update;
```

Build output

```
show_vector_update.adb:3:09: warning: no entities of "Ada.Text_IO" are referenced
  [-gnatwu]
show_vector_update.adb:3:19: warning: use clause for package "Text_IO" has no
  effect [-gnatwu]
```

19.1.6 Finding and changing elements

You can locate a specific element in a vector by retrieving its index. `Find_Index` retrieves the index of the first element matching the value you're looking for. Alternatively, you can use `Find` to retrieve a cursor referencing that element. For example:

Listing 10: `show_find_vector_element.adb`

```

1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Find_Vector_Element is
6
7    package Integer_Vectors is new
8      Ada.Containers.Vectors
9      (Index_Type  => Natural,
10       Element_Type => Integer);
11
12   use Integer_Vectors;
13
14   V : Vector := 20 & 10 & 0 & 13;
15   Idx : Extended_Index;
16   C : Cursor;
17 begin
18   -- Using Find_Index to retrieve the index
19   -- of element with value 10
20   Idx := V.Find_Index (10);
21   Put_Line ("Index of element with value 10 is "
22             & Extended_Index'Image (Idx));
23
24   -- Using Find to retrieve the cursor for
25   -- the element with value 13
26   C := V.Find (13);
27   Idx := To_Index (C);
28   Put_Line ("Index of element with value 13 is "
29             & Extended_Index'Image (Idx));
30 end Show_Find_Vector_Element;

```

Build output

```
show_find_vector_element.adb:14:04: warning: "V" is not modified, could be
→declared constant [-gnatwk]
```

Runtime output

```
Index of element with value 10 is 1
Index of element with value 13 is 3
```

As we saw in the previous section, we can directly access vector elements by using either an index or cursor. However, an exception is raised if we try to access an element with an invalid index or cursor, so we must check whether the index or cursor is valid before using it to access an element. In our example, `Find_Index` or `Find` might not have found the element in the vector. We check for this possibility by comparing the index to `No_Index` or the cursor to `No_Element`. For example:

```
-- Modify vector element using index
if Idx /= No_Index then
  V (Idx) := 11;
end if;
```

(continues on next page)

(continued from previous page)

```
-- Modify vector element using cursor
if C /= No_Element then
  V (C) := 14;
end if;
```

Instead of writing `V (C) := 14`, we could use the longer form `V.Replace_Element (C, 14)`.

19.1.7 Inserting elements

In the previous sections, we've seen examples of how to add elements to a vector:

- using the concatenation operator (`&`) at the vector declaration, or
- calling the `Prepend` and `Append` procedures.

You may want to insert an element at a specific position, e.g. before a certain element in the vector. You do this by calling `Insert`. For example:

Listing 11: show_vector_insert.adb

```
1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Insert is
7
8    package Integer_Vectors is new
9      Ada.Containers.Vectors
10     (Index_Type  => Natural,
11      Element_Type => Integer);
12
13  use Integer_Vectors;
14
15  procedure Show_Elements (V : Vector) is
16  begin
17    New_Line;
18    Put_Line ("Vector has "
19              & Count_Type'Image (V.Length)
20              & " elements");
21
22    if not V.Is_Empty then
23      Put_Line ("Vector elements are: ");
24      for E of V loop
25        Put_Line ("- " & Integer'Image (E));
26      end loop;
27    end if;
28  end Show_Elements;
29
30  V : Vector := 20 & 10 & 12;
31  C : Cursor;
32
33 begin
34  Show_Elements (V);
35
36  New_Line;
37  Put_Line ("Adding element with value 9 (before 10)...");
38
39  -- -- Using V.Insert to insert the element
```

(continues on next page)

(continued from previous page)

```

40   -- into the vector
41   --
42   C := V.Find (10);
43   if C /= No_Element then
44     V.Insert (C, 9);
45   end if;
46
47   Show_Elements (V);
48
49 end Show_Vector_Insert;

```

Runtime output

```

Vector has 3 elements
Vector elements are:
- 20
- 10
- 12

Adding element with value 9 (before 10)...

Vector has 4 elements
Vector elements are:
- 20
- 9
- 10
- 12

```

In this example, we're looking for an element with the value of 10. If we find it, we insert an element with the value of 9 before it.

19.1.8 Removing elements

You can remove elements from a vector by passing either a valid index or cursor to the Delete procedure. If we combine this with the functions Find_Index and Find from the previous section, we can write a program that searches for a specific element and deletes it, if found:

Listing 12: show_remove_vector_element.adb

```

1  with Ada.Containers.Vectors;
2
3  with Ada.Text_IO; use Ada.Text_IO;
4
5  procedure Show_Remove_Vector_Element is
6    package Integer_Vectors is new
7      Ada.Containers.Vectors
8        (Index_Type  => Natural,
9         Element_Type => Integer);
10
11  use Integer_Vectors;
12
13  V : Vector := 20 & 10 & 0 & 13 & 10 & 13;
14  Idx : Extended_Index;
15  C   : Cursor;
16 begin
17   -- Use Find_Index to retrieve index of
18   -- the element with value 10

```

(continues on next page)

(continued from previous page)

```

19  Idx := V.Find_Index (10);
20
21  -- Checking whether index is valid
22  if Idx /= No_Index then
23      -- Removing element using V.Delete
24      V.Delete (Idx);
25  end if;
26
27  -- Use Find to retrieve cursor for
28  -- the element with value 13
29  C := V.Find (13);
30
31  -- Check whether index is valid
32  if C /= No_Element then
33      -- Remove element using V.Delete
34      V.Delete (C);
35  end if;
36
37 end Show_Remove_Vector_Element;

```

Build output

```

show_remove_vector_element.adb:3:09: warning: no entities of "Ada.Text_Io" are
→referenced [-gnatwu]
show_remove_vector_element.adb:3:19: warning: use clause for package "Text_Io" has
→no effect [-gnatwu]

```

We can extend this approach to delete all elements matching a certain value. We just need to keep searching for the element in a loop until we get an invalid index or cursor. For example:

Listing 13: show_remove_vector_elements.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_Io; use Ada.Text_Io;
5
6  procedure Show_Remove_Vector_Elements is
7
8      package Integer_Vectors is new
9          Ada.Containers.Vectors
10         (Index_Type    => Natural,
11          Element_Type => Integer);
12
13  use Integer_Vectors;
14
15  procedure Show_Elements (V : Vector) is
16  begin
17      New_Line;
18      Put_Line ("Vector has "
19                 & Count_Type'Image (V.Length)
20                 & " elements");
21
22      if not V.Is_Empty then
23          Put_Line ("Vector elements are: ");
24          for E of V loop
25              Put_Line ("- " & Integer'Image (E));
26          end loop;
27      end if;
28  end Show_Elements;

```

(continues on next page)

(continued from previous page)

```
29      V : Vector := 20 & 10 & 0 & 13 & 10 & 14 & 13;
30
31  begin
32    Show_Elements (V);
33
34    -- Remove elements using an index
35    --
36    declare
37      E : constant Integer := 10;
38      I : Extended_Index;
39    begin
40      New_Line;
41      Put_Line ("Removing all elements with value of "
42                 & Integer'Image (E) & "...");
43      loop
44        I := V.Find_Index (E);
45        exit when I = No_Index;
46        V.Delete (I);
47      end loop;
48    end;
49
50    --
51    -- Remove elements using a cursor
52    --
53    declare
54      E : constant Integer := 13;
55      C : Cursor;
56    begin
57      New_Line;
58      Put_Line ("Removing all elements with value of "
59                 & Integer'Image (E) & "...");
60      loop
61        C := V.Find (E);
62        exit when C = No_Element;
63        V.Delete (C);
64      end loop;
65    end;
66
67    Show_Elements (V);
68  end Show_Remove_Vector_Elements;
```

Runtime output

```
Vector has 7 elements
Vector elements are:
- 20
- 10
- 0
- 13
- 10
- 14
- 13

Removing all elements with value of 10...

Removing all elements with value of 13...

Vector has 3 elements
Vector elements are:
```

(continues on next page)

(continued from previous page)

```
- 20
- 0
- 14
```

In this example, we remove all elements with the value 10 from the vector by retrieving their index. Likewise, we remove all elements with the value 13 by retrieving their cursor.

19.1.9 Other Operations

We've seen some operations on vector elements. Here, we'll see operations on the vector as a whole. The most prominent is the concatenation of multiple vectors, but we'll also see operations on vectors, such as sorting and sorted merging operations, that view the vector as a sequence of elements and operate on the vector considering the element's relations to each other.

We do vector concatenation using the `&` operator on vectors. Let's consider two vectors `V1` and `V2`. We can concatenate them by doing `V := V1 & V2`. `V` contains the resulting vector.

The generic package `Generic_Sorting` is a child package of `Ada.Containers.Vectors`. It contains sorting and merging operations. Because it's a generic package, you can't use it directly, but have to instantiate it. In order to use these operations on a vector of integer values (`Integer_Vectors`, in our example), you need to instantiate it directly as a child of `Integer_Vectors`. The next example makes it clear how to do this.

After instantiating `Generic_Sorting`, we make all the operations available to us with the `use` statement. We can then call `Sort` to sort the vector and `Merge` to merge one vector into another.

The following example presents code that manipulates three vectors (`V1`, `V2`, `V3`) using the concatenation, sorting and merging operations:

Listing 14: show_vector_ops.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Vectors;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Vector_Ops is
7
8    package Integer_Vectors is new
9      Ada.Containers.Vectors
10     (Index_Type    => Natural,
11      Element_Type => Integer);
12
13  package Integer_Vectors_Sorting is new Integer_Vectors.Generic_Sorting;
14
15  use Integer_Vectors;
16  use Integer_Vectors_Sorting;
17
18  procedure Show_Elements (V : Vector) is
19  begin
20    New_Line;
21    Put_Line ("Vector has "
22              & Count_Type'Image (V.Length)
23              & " elements");
24
25    if not V.Is_Empty then
26      Put_Line ("Vector elements are: ");

```

(continues on next page)

(continued from previous page)

```
27      for E of V loop
28          Put_Line (" - " & Integer'Image (E));
29      end loop;
30  end if;
31 end Show_Elements;

32
33 V, V1, V2, V3 : Vector;
34 begin
35     V1 := 10 & 12 & 18;
36     V2 := 11 & 13 & 19;
37     V3 := 15 & 19;
38
39     New_Line;
40     Put_Line ("----- V1 -----");
41     Show_Elements (V1);
42
43     New_Line;
44     Put_Line ("----- V2 -----");
45     Show_Elements (V2);
46
47     New_Line;
48     Put_Line ("----- V3 -----");
49     Show_Elements (V3);
50
51     New_Line;
52     Put_Line ("Concatenating V1, V2 and V3 into V:");
53
54     V := V1 & V2 & V3;
55
56     Show_Elements (V);
57
58     New_Line;
59     Put_Line ("Sorting V:");
60
61     Sort (V);
62
63     Show_Elements (V);
64
65     New_Line;
66     Put_Line ("Merging V2 into V1:");
67
68     Merge (V1, V2);
69
70     Show_Elements (V1);
71
72 end Show_Vector_Ops;
```

Runtime output

```
----- V1 -----
Vector has 3 elements
Vector elements are:
- 10
- 12
- 18
----- V2 -----
Vector has 3 elements
```

(continues on next page)

(continued from previous page)

Vector elements are:

- 11
- 13
- 19

---- V3 ----

Vector has 2 elements

Vector elements are:

- 15
- 19

Concatenating V1, V2 and V3 into V:

Vector has 8 elements

Vector elements are:

- 10
- 12
- 18
- 11
- 13
- 19
- 15
- 19

Sorting V:

Vector has 8 elements

Vector elements are:

- 10
- 11
- 12
- 13
- 15
- 18
- 19
- 19

Merging V2 into V1:

Vector has 6 elements

Vector elements are:

- 10
- 11
- 12
- 13
- 18
- 19

The Reference Manual requires that the worst-case complexity of a call to Sort be $O(N^2)$ and the average complexity be better than $O(N^2)$.

19.2 Sets

Sets are another class of containers. While vectors allow duplicated elements to be inserted, sets ensure that no duplicated elements exist.

In the following sections, we'll see operations you can perform on sets. However, since many of the operations on vectors are similar to the ones used for sets, we'll cover them more quickly here. Please refer back to the section on vectors for a more detailed discussion.

19.2.1 Initialization and iteration

To initialize a set, you can call the `Insert` procedure. However, if you do, you need to ensure no duplicate elements are being inserted: if you try to insert a duplicate, you'll get an exception. If you have less control over the elements to be inserted so that there may be duplicates, you can use another option instead:

- a version of `Insert` that returns a Boolean value indicating whether the insertion was successful;
- the `Include` procedure, which silently ignores any attempt to insert a duplicated element.

To iterate over a set, you can use a `for E of S loop`, as you saw for vectors. This gives you a reference to each element in the set.

Let's see an example:

Listing 15: show_set_init.adb

```
1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Ordered_Sets;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Set_Init is
7
8    package Integer_Sets is new
9      Ada.Containers.Ordered_Sets
10     (Element_Type => Integer);
11
12   use Integer_Sets;
13
14   S : Set;
15   -- Same as: S : Integer_Sets.Set;
16   C : Cursor;
17   Ins : Boolean;
18 begin
19   S.Insert (20);
20   S.Insert (10);
21   S.Insert (0);
22   S.Insert (13);
23
24   -- Calling S.Insert(0) now would raise
25   -- Constraint_Error because this element
26   -- is already in the set. We instead call a
27   -- version of Insert that doesn't raise an
28   -- exception but instead returns a Boolean
29   -- indicating the status
```

(continues on next page)

(continued from previous page)

```

31  S.Insert (0, C, Ins);
32  if not Ins then
33      Put_Line ("Inserting 0 into set was not successful");
34  end if;
35
36  -- We can also call S.Include instead
37  -- If the element is already present,
38  -- the set remains unchanged
39  S.Include (0);
40  S.Include (13);
41  S.Include (14);
42
43  Put_Line ("Set has "
44      & Count_Type'Image (S.Length)
45      & " elements");
46
47  --
48  -- Iterate over set using for .. of loop
49  --
50  Put_Line ("Elements:");
51  for E of S loop
52      Put_Line ("- " & Integer'Image (E));
53  end loop;
54 end Show_Set_Init;

```

Runtime output

```

Inserting 0 into set was not successful
Set has 5 elements
Elements:
- 0
- 10
- 13
- 14
- 20

```

19.2.2 Operations on elements

In this section, we briefly explore the following operations on sets:

- Delete and Exclude to remove elements;
- Contains and Find to verify the existence of elements.

To delete elements, you call the procedure `Delete`. However, analogously to the `Insert` procedure above, `Delete` raises an exception if the element to be deleted isn't present in the set. If you want to permit the case where an element might not exist, you can call `Exclude`, which silently ignores any attempt to delete a non-existent element.

`Contains` returns a Boolean value indicating whether a value is contained in the set. `Find` also looks for an element in a set, but returns a cursor to the element or `No_Element` if the element doesn't exist. You can use either function to search for elements in a set.

Let's look at an example that makes use of these operations:

Listing 16: `show_set_element_ops.adb`

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Ordered_Sets;
3

```

(continues on next page)

(continued from previous page)

```

4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Set_Element_Ops is
7
8    package Integer_Sets is new
9      Ada.Containers.Ordered_Sets
10     (Element_Type => Integer);
11
12   use Integer_Sets;
13
14   procedure Show_Elements (S : Set) is
15   begin
16     New_Line;
17     Put_Line ("Set has "
18               & Count_Type'Image (S.Length)
19               & " elements");
20     Put_Line ("Elements:");
21     for E of S loop
22       Put_Line ("- " & Integer'Image (E));
23     end loop;
24   end Show_Elements;
25
26   S : Set;
27 begin
28   S.Insert (20);
29   S.Insert (10);
30   S.Insert (0);
31   S.Insert (13);
32
33   S.Delete (13);
34
35   -- Calling S.Delete (13) again raises
36   -- Constraint_Error because the element
37   -- is no longer present in the set, so
38   -- it can't be deleted. We can call
39   -- V.Exclude instead:
40   S.Exclude (13);
41
42   if S.Contains (20) then
43     Put_Line ("Found element 20 in set");
44   end if;
45
46   -- Alternatively, we could use S.Find
47   -- instead of S.Contains
48   if S.Find (0) /= No_Element then
49     Put_Line ("Found element 0 in set");
50   end if;
51
52   Show_Elements (S);
53 end Show_Set_Element_Ops;

```

Runtime output

```
Found element 20 in set
Found element 0 in set
```

```
Set has 3 elements
Elements:
- 0
- 10
- 20
```

In addition to ordered sets used in the examples above, the standard library also offers hashed sets. The Reference Manual requires the following average complexity of each operation:

Operations	Ordered_Sets	Hashed_Sets
<ul style="list-style-type: none"> • Insert • Include • Replace • Delete • Exclude • Find 	$O((\log N)^2)$ or better	$O(\log N)$
Subprogram using cursor	$O(1)$	$O(1)$

19.2.3 Other Operations

The previous sections mostly dealt with operations on individual elements of a set. But Ada also provides typical set operations: union, intersection, difference and symmetric difference. In contrast to some vector operations we've seen before (e.g. Merge), here you can use built-in operators, such as `-`. The following table lists the operations and its associated operator:

Set Operation	Operator
Union	<code>or</code>
Intersection	<code>and</code>
Difference	<code>-</code>
Symmetric difference	<code>xor</code>

The following example makes use of these operators:

Listing 17: show_set_ops.adb

```

1  with Ada.Containers; use Ada.Containers;
2  with Ada.Containers.Ordered_Sets;
3
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  procedure Show_Set_Ops is
7
8    package Integer_Sets is new
9      Ada.Containers.Ordered_Sets
10     (Element_Type => Integer);
11
12   use Integer_Sets;
13
14   procedure Show_Elements (S : Set) is
15   begin
16     Put_Line ("Elements:");
17     for E of S loop
18       Put_Line ("- " & Integer'Image (E));
19     end loop;
20   end Show_Elements;
21
22   procedure Show_Op (S          : Set;
23                      Op_Name : String) is
24   begin
25     New_Line;

```

(continues on next page)

(continued from previous page)

```
26      Put_Line (Op_Name
27          & "(set #1, set #2) has "
28          & Count_Type'Image (S.Length)
29          & " elements");
30  end Show_Op;
31
32  S1, S2, S3 : Set;
33 begin
34  S1.Insert (0);
35  S1.Insert (10);
36  S1.Insert (13);
37
38  S2.Insert (0);
39  S2.Insert (10);
40  S2.Insert (14);
41
42  S3.Insert (0);
43  S3.Insert (10);
44
45  New_Line;
46  Put_Line ("---- Set #1 ----");
47  Show_Elements (S1);
48
49  New_Line;
50  Put_Line ("---- Set #2 ----");
51  Show_Elements (S2);
52
53  New_Line;
54  Put_Line ("---- Set #3 ----");
55  Show_Elements (S3);
56
57  New_Line;
58  if S3.Is_Subset (S1) then
59      Put_Line ("S3 is a subset of S1");
60  else
61      Put_Line ("S3 is not a subset of S1");
62  end if;
63
64  S3 := S1 and S2;
65  Show_Op (S3, "Intersection");
66  Show_Elements (S3);
67
68  S3 := S1 or S2;
69  Show_Op (S3, "Union");
70  Show_Elements (S3);
71
72  S3 := S1 - S2;
73  Show_Op (S3, "Difference");
74  Show_Elements (S3);
75
76  S3 := S1 xor S2;
77  Show_Op (S3, "Symmetric difference");
78  Show_Elements (S3);
79
80 end Show_Set_Ops;
```

Runtime output

```
---- Set #1 ----
Elements:
```

(continues on next page)

(continued from previous page)

```

- 0
- 10
- 13

---- Set #2 ----
Elements:
- 0
- 10
- 14

---- Set #3 ----
Elements:
- 0
- 10

S3 is a subset of S1

Intersection(set #1, set #2) has 2 elements
Elements:
- 0
- 10

Union(set #1, set #2) has 4 elements
Elements:
- 0
- 10
- 13
- 14

Difference(set #1, set #2) has 1 elements
Elements:
- 13

Symmetric difference(set #1, set #2) has 2 elements
Elements:
- 13
- 14

```

19.3 Indefinite maps

The previous sections presented containers for elements of definite types. Although most examples in those sections presented **Integer** types as element type of the containers, containers can also be used with indefinite types, an example of which is the **String** type. However, indefinite types require a different kind of containers designed specially for them.

We'll also be exploring a different class of containers: maps. They associate a key with a specific value. An example of a map is the one-to-one association between a person and their age. If we consider a person's name to be the key, the value is the person's age.

19.3.1 Hashed maps

Hashed maps are maps that make use of a hash as a key. The hash itself is calculated by a function you provide.

In other languages

Hashed maps are similar to dictionaries in Python and hashes in Perl. One of the main differences is that these scripting languages allow using different types for the values contained in a single map, while in Ada, both the type of key and value are specified in the package instantiation and remains constant for that specific map. You can't have a map where two elements are of different types or two keys are of different types. If you want to use multiple types, you must create a different map for each and use only one type in each map.

When instantiating a hashed map from `Ada.Containers.Indefinite_Hashed_Maps`, we specify following elements:

- `Key_Type`: type of the key
- `Element_Type`: type of the element
- `Hash`: hash function for the `Key_Type`
- `Equivalent_Keys`: an equality operator (e.g. `=`) that indicates whether two keys are to be considered equal.
 - If the type specified in `Key_Type` has a standard operator, you can use it, which you do by specifying that operator as the value of `Equivalent_Keys`.

In the next example, we'll use a string as a key type. We'll use the `Hash` function provided by the standard library for strings (in the `Ada.Strings` package) and the standard equality operator.

You add elements to a hashed map by calling `Insert`. If an element is already contained in a map `M`, you can access it directly by using its key. For example, you can change the value of an element by calling `M ("My_Key") := 10`. If the key is not found, an exception is raised. To verify if a key is available, use the function `Contains` (as we've seen above in the section on sets).

Let's see an example:

Listing 18: `show_hashed_map.adb`

```
1 with Ada.Containers.Indefinite_Hashed_Maps;
2 with Ada.Strings.Hash;
3
4 with Ada.Text_IO; use Ada.Text_IO;
5
6 procedure Show_Hashed_Map is
7
8   package Integer_Hashed_Maps is new
9     Ada.Containers.Indefinite_Hashed_Maps
10    (Key_Type      => String,
11     Element_Type  => Integer,
12     Hash          => Ada.Strings.Hash,
13     Equivalent_Keys => "=");
14
15   use Integer_Hashed_Maps;
16
17   M : Map;
18   -- Same as:
```

(continues on next page)

(continued from previous page)

```

19   --
20   -- M : Integer_Hashed_Maps.Map;
21 begin
22   M.Include ("Alice", 24);
23   M.Include ("John", 40);
24   M.Include ("Bob", 28);
25
26   if M.Contains ("Alice") then
27     Put_Line ("Alice's age is "
28             & Integer'Image (M ("Alice")));
29   end if;
30
31   -- Update Alice's age
32   -- Key must already exist in M.
33   -- Otherwise an exception is raised.
34   M ("Alice") := 25;
35
36   New_Line; Put_Line ("Name & Age:");
37   for C in M.Iterate loop
38     Put_Line (Key (C) & ": "
39             & Integer'Image (M (C)));
40   end loop;
41
42 end Show_Hashed_Map;

```

Runtime output

```
Alice's age is 24
```

```
Name & Age:  
John: 40  
Bob: 28  
Alice: 25
```

19.3.2 Ordered maps

Ordered maps share many features with hashed maps. The main differences are:

- A hash function isn't needed. Instead, you must provide an ordering function (< operator), which the ordered map will use to order elements and allow fast access, O(log N), using a binary search.
 - If the type specified in Key_Type has a standard < operator, you can use it in a similar way as we did for Equivalent_Keys above for hashed maps.

Let's see an example:

Listing 19: show_ordered_map.adb

```

1 with Ada.Containers.Indefinite_Ordered_Maps;
2
3 with Ada.Text_IO; use Ada.Text_IO;
4
5 procedure Show_Ordered_Map is
6
7   package Integer_Ordered_Maps is new
8     Ada.Containers.Indefinite_Ordered_Maps
9     (Key_Type      => String,
10      Element_Type => Integer);
11

```

(continues on next page)

(continued from previous page)

```

12  use Integer_Ordered_Maps;
13
14  M : Map;
15 begin
16  M.Include ("Alice", 24);
17  M.Include ("John", 40);
18  M.Include ("Bob", 28);
19
20  if M.Contains ("Alice") then
21    Put_Line ("Alice's age is "
22              & Integer'Image (M ("Alice")));
23  end if;
24
25  -- Update Alice's age
26  -- Key must already exist in M
27  M ("Alice") := 25;
28
29  New_Line; Put_Line ("Name & Age:");
30  for C in M.Iterate loop
31    Put_Line (Key (C) & ": "
32              & Integer'Image (M (C)));
33  end loop;
34
35 end Show_Ordered_Map;

```

Runtime output

```
Alice's age is 24
```

```
Name & Age:
Alice: 25
Bob: 28
John: 40
```

You can see a great similarity between the examples above and from the previous section. In fact, since both kinds of maps share many operations, we didn't need to make extensive modifications when we changed our example to use ordered maps instead of hashed maps. The main difference is seen when we run the examples: the output of a hashed map is usually unordered, but the output of a ordered map is always ordered, as implied by its name.

19.3.3 Complexity

Hashed maps are generally the fastest data structure available to you in Ada if you need to associate heterogeneous keys to values and search for them quickly. In most cases, they are slightly faster than ordered maps. So if you don't need ordering, use hashed maps.

The Reference Manual requires the following average complexity of operations:

Operations	Ordered_Maps	Hashed_Maps
<ul style="list-style-type: none"> • Insert • Include • Replace • Delete • Exclude • Find 	$O((\log N)^2)$ or better	$O(\log N)$
Subprogram using cursor	$O(1)$	$O(1)$

STANDARD LIBRARY: DATES & TIMES

The standard library supports processing of dates and times using two approaches:

- *Calendar* approach, which is suitable for handling dates and times in general;
- *Real-time* approach, which is better suited for real-time applications that require enhanced precision — for example, by having access to an absolute clock and handling time spans. Note that this approach only supports times, not dates.

The following sections present these two approaches.

20.1 Date and time handling

The Ada.Calendar package supports handling of dates and times. Let's look at a simple example:

Listing 1: display_current_time.adb

```
1 with Ada.Calendar;           use Ada.Calendar;
2 with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3 with Ada.Text_IO;            use Ada.Text_IO;
4
5 procedure Display_Current_Time is
6   Now : Time := Clock;
7 begin
8   Put_Line ("Current time: " & Image (Now));
9 end Display_Current_Time;
```

Build output

```
display_current_time.adb:6:04: warning: "Now" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Current time: 2023-01-28 01:45:39
```

This example displays the current date and time, which is retrieved by a call to the `Clock` function. We call the function `Image` from the `Ada.Calendar.Formatting` package to get a **String** for the current date and time. We could instead retrieve each component using the `Split` function. For example:

Listing 2: display_current_year.adb

```
1 with Ada.Calendar;           use Ada.Calendar;
2 with Ada.Text_IO;            use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```

4  procedure Display_Current_Year is
5      Now          : Time := Clock;
6
7      Now_Year     : Year_Number;
8      Now_Month   : Month_Number;
9      Now_Day      : Day_Number;
10     Now_Seconds : Day_Duration;
11
12 begin
13     Split (Now,
14         Now_Year,
15         Now_Month,
16         Now_Day,
17         Now_Seconds);
18
19     Put_Line ("Current year is: "
20             & Year_Number'Image (Now_Year));
21     Put_Line ("Current month is: "
22             & Month_Number'Image (Now_Month));
23     Put_Line ("Current day   is: "
24             & Day_Number'Image (Now_Day));
25 end Display_Current_Year;

```

Build output

```
display_current_year.adb:5:04: warning: "Now" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Current year is: 2023
Current month is: 1
Current day   is: 28
```

Here, we're retrieving each element and displaying it separately.

20.1.1 Delaying using date

You can delay an application so that it restarts at a specific date and time. We saw something similar in the chapter on tasking. You do this using a **delay until** statement. For example:

Listing 3: display_delay_next_specific_time.adb

```

1  with Ada.Calendar;           use Ada.Calendar;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3  with Ada.Calendar.Time_Zones; use Ada.Calendar.Time_Zones;
4  with Ada.Text_IO;            use Ada.Text_IO;
5
6  procedure Display_Delay_Next_Specific_Time is
7      TZ   : Time_Offset := UTC_Time_Offset;
8      Next : Time        :=
9          Ada.Calendar.Formatting.Time_Of
10         (Year      => 2018,
11          Month    => 5,
12          Day      => 1,
13          Hour     => 15,
14          Minute   => 0,
15          Second   => 0,
16          Sub_Second => 0.0,

```

(continues on next page)

(continued from previous page)

```

17     Leap_Second => False,
18     Time_Zone    => TZ);
19
20     -- Next = 2018-05-01 15:00:00.00
21     --          (local time-zone)
22 begin
23     Put_Line ("Let's wait until...");  

24     Put_Line (Image (Next, True, TZ));
25
26     delay until Next;
27
28     Put_Line ("Enough waiting!");
29 end Display_Delay_Next_Specific_Time;

```

Build output

```

display_delay_next_specific_time.adb:7:04: warning: "TZ" is not modified, could be_
→ declared constant [-gnatwk]
display_delay_next_specific_time.adb:8:04: warning: "Next" is not modified, could_
→ be declared constant [-gnatwk]

```

Runtime output

```

Let's wait until...
2018-05-01 15:00:00.00
Enough waiting!

```

In this example, we specify the date and time by initializing `Next` using a call to `Time_0f`, a function taking the various components of a date (year, month, etc) and returning an element of the `Time` type. Because the date specified is in the past, the `delay until` statement won't produce any noticeable effect. However, if we passed a date in the future, the program would wait until that specific date and time arrived.

Here we're converting the time to the local timezone. If we don't specify a timezone, *Coordinated Universal Time* (abbreviated to UTC) is used by default. By retrieving the time offset to UTC with a call to `UTC_Time_Offset` from the `Ada.Calendar.Time_Zones` package, we can initialize `TZ` and use it in the call to `Time_0f`. This is all we need do to make the information provided to `Time_0f` relative to the local time zone.

We could achieve a similar result by initializing `Next` with a `String`. We can do this with a call to `Value` from the `Ada.Calendar.Formatting` package. This is the modified code:

Listing 4: `display_delay_next_specific_time.adb`

```

1  with Ada.Calendar;           use Ada.Calendar;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3  with Ada.Calendar.Time_Zones; use Ada.Calendar.Time_Zones;
4  with Ada.Text_IO;           use Ada.Text_IO;
5
6 procedure Display_Delay_Next_Specific_Time is
7   TZ   : Time_Offset := UTC_Time_Offset;
8   Next : Time      :=
9     Ada.Calendar.Formatting.Value
10    ("2018-05-01 15:00:00.00", TZ);
11
12   -- Next = 2018-05-01 15:00:00.00
13   --          (local time-zone)
14 begin
15   Put_Line ("Let's wait until...");  

16   Put_Line (Image (Next, True, TZ));
17

```

(continues on next page)

(continued from previous page)

```

18  delay until Next;
19
20  Put_Line ("Enough waiting!");
21 end Display_Delay_Next_Specific_Time;
```

Build output

```
display_delay_next_specific_time.adb:7:04: warning: "TZ" is not modified, could be declared constant [-gnatwk]
display_delay_next_specific_time.adb:8:04: warning: "Next" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Let's wait until...
2018-05-01 15:00:00.00
Enough waiting!
```

In this example, we're again using TZ in the call to Value to adjust the input time to the current time zone.

In the examples above, we were delaying to a specific date and time. Just like we saw in the tasking chapter, we could instead specify the delay relative to the current time. For example, we could delay by 5 seconds, using the current time:

Listing 5: display_delay_next.adb

```

1  with Ada.Calendar;           use Ada.Calendar;
2  with Ada.Text_IO;            use Ada.Text_IO;
3
4  procedure Display_Delay_Next is
5    D : Duration := 5.0;
6    --          ^ seconds
7    Now : Time     := Clock;
8    Next : Time     := Now + D;
9    --          ^ use duration to
10   --          specify next point
11   --          in time
12 begin
13   Put_Line ("Let's wait "
14         & Duration'Image (D) & " seconds...");
15   delay until Next;
16   Put_Line ("Enough waiting!");
17 end Display_Delay_Next;
```

Build output

```
display_delay_next.adb:5:04: warning: "D" is not modified, could be declared constant [-gnatwk]
display_delay_next.adb:7:04: warning: "Now" is not modified, could be declared constant [-gnatwk]
display_delay_next.adb:8:04: warning: "Next" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
Let's wait 5.000000000 seconds...
Enough waiting!
```

Here, we're specifying a duration of 5 seconds in D, adding it to the current time from Now, and storing the sum in Next. We then use it in the `delay until` statement.

20.2 Real-time

In addition to Ada.Calendar, the standard library also supports time operations for real-time applications. These are included in the Ada.Real_Time package. This package also include a Time type. However, in the Ada.Real_Time package, the Time type is used to represent an absolute clock and handle a time span. This contrasts with the Ada.Calendar, which uses the Time type to represent dates and times.

In the previous section, we used the Time type from the Ada.Calendar and the **delay until** statement to delay an application by 5 seconds. We could have used the Ada.Real_Time package instead. Let's modify that example:

Listing 6: display_delay_next_real_time.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Ada.Real_Time;    use Ada.Real_Time;
3
4  procedure Display_Delay_Next_Real_Time is
5      D      : Time_Span := Seconds (5);
6      Next   : Time      := Clock + D;
7  begin
8      Put_Line ("Let's wait "
9                 & Duration'Image (To_Duration (D))
10                & " seconds...");
11      delay until Next;
12      Put_Line ("Enough waiting!");
13  end Display_Delay_Next_Real_Time;

```

Build output

```

display_delay_next_real_time.adb:5:04: warning: "D" is not modified, could be u
→ declared constant [-gnatwk]
display_delay_next_real_time.adb:6:04: warning: "Next" is not modified, could be u
→ declared constant [-gnatwk]

```

Runtime output

```

Let's wait 5.000000000 seconds...
Enough waiting!

```

The main difference is that D is now a variable of type Time_Span, defined in the Ada.Real_Time package. We call the function Seconds to initialize D, but could have gotten a finer granularity by calling Nanoseconds instead. Also, we need to first convert D to the Duration type using To_Duration before we can display it.

20.2.1 Benchmarking

One interesting application using the Ada.Real_Time package is benchmarking. We've used that package before in a previous section when discussing tasking. Let's look at an example of benchmarking:

Listing 7: display_benchmarking.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Ada.Real_Time;    use Ada.Real_Time;
3
4  procedure Display_Benchmarking is
5
6      procedure Computational_Intensive_App is

```

(continues on next page)

(continued from previous page)

```

7   begin
8     delay 5.0;
9   end Computational_Intensive_App;
10
11  Start_Time, Stop_Time : Time;
12  Elapsed_Time          : Time_Span;
13
14 begin
15   Start_Time := Clock;
16
17   Computational_Intensive_App;
18
19   Stop_Time    := Clock;
20   Elapsed_Time := Stop_Time - Start_Time;
21
22   Put_Line ("Elapsed time: "
23             & Duration'Image (To_Duration (Elapsed_Time))
24             & " seconds");
25 end Display_Benchmarking;

```

Runtime output

```
Elapsed time: 5.000112411 seconds
```

This example defines a dummy Computational_Intensive_App implemented using a simple `delay` statement. We initialize Start_Time and Stop_Time from the then-current clock and calculate the elapsed time. By running this program, we see that the time is roughly 5 seconds, which is expected due to the `delay` statement.

A similar application is benchmarking of CPU time. We can implement this using the Execution_Time package. Let's modify the previous example to measure CPU time:

Listing 8: display_benchmarking_cpu_time.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Real_Time;         use Ada.Real_Time;
3  with Ada.Execution_Time;   use Ada.Execution_Time;
4
5  procedure Display_Benchmarking_CPU_Time is
6
7    procedure Computational_Intensive_App is
8    begin
9      delay 5.0;
10   end Computational_Intensive_App;
11
12  Start_Time, Stop_Time : CPU_Time;
13  Elapsed_Time          : Time_Span;
14
15 begin
16   Start_Time := Clock;
17
18   Computational_Intensive_App;
19
20   Stop_Time    := Clock;
21   Elapsed_Time := Stop_Time - Start_Time;
22
23   Put_Line ("CPU time: "
24             & Duration'Image (To_Duration (Elapsed_Time))
25             & " seconds");
26 end Display_Benchmarking_CPU_Time;

```

Runtime output

```
CPU time: 0.000089047 seconds
```

In this example, `Start_Time` and `Stop_Time` are of type `CPU_Time` instead of `Time`. However, we still call the `Clock` function to initialize both variables and calculate the elapsed time in the same way as before. By running this program, we see that the CPU time is significantly lower than the 5 seconds we've seen before. This is because the `delay` statement doesn't require much CPU time. The results will be different if we change the implementation of `Computational_Intensive_App` to use a mathematical functions in a long loop. For example:

Listing 9: `display_benchmarking_math.adb`

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Real_Time;         use Ada.Real_Time;
3  with Ada.Execution_Time;   use Ada.Execution_Time;
4
5  with Ada.Numerics.Generic_Elementary_Functions;
6
7  procedure Display_Benchmarking_Math is
8
9    procedure Computational_Intensive_App is
10      package Funcs is new Ada.Numerics.Generic_Elementary_Functions
11        (Float_Type => Long_Long_Float);
12      use Funcs;
13
14      X : Long_Long_Float;
15    begin
16      for I in 0 .. 1_000_000 loop
17        X := Tan (Arctan
18                  (Tan (Arctan
19                      (Tan (Arctan
20                          (Tan (Arctan
21                            (Tan (Arctan
22                              (Tan (Arctan
23                                (0.577))))))))));
24    end loop;
25  end Computational_Intensive_App;
26
27  procedure Benchm_Elapsed_Time is
28    Start_Time, Stop_Time : Time;
29    Elapsed_Time          : Time_Span;
30
31  begin
32    Start_Time := Clock;
33
34    Computational_Intensive_App;
35
36    Stop_Time    := Clock;
37    Elapsed_Time := Stop_Time - Start_Time;
38
39    Put_Line ("Elapsed time: "
40              & Duration'Image (To_Duration (Elapsed_Time))
41              & " seconds");
42  end Benchm_Elapsed_Time;
43
44  procedure Benchm_CPU_Time is
45    Start_Time, Stop_Time : CPU_Time;
46    Elapsed_Time          : Time_Span;
47
48  begin
49    Start_Time := Clock;

```

(continues on next page)

(continued from previous page)

```
50      Computational_Intensive_App;
51
52      Stop_Time    := Clock;
53      Elapsed_Time := Stop_Time - Start_Time;
54
55      Put_Line ("CPU time: "
56                  & Duration'Image (To_Duration (Elapsed_Time))
57                  & " seconds");
58
59  end Benchm_CPU_Time;
60 begin
61   Benchm_Elapsed_Time;
62   Benchm_CPU_Time;
63 end Display_Benchmarking_Math;
```

Build output

```
display_benchmarking_math.adb:14:07: warning: variable "X" is assigned but never ↴
read [-gnatwm]
```

Runtime output

```
Elapsed time: 1.355411464 seconds
CPU time: 1.264107549 seconds
```

Now that our dummy `Computational_Intensive_App` involves mathematical operations requiring significant CPU time, the measured elapsed and CPU time are much closer to each other than before.

STANDARD LIBRARY: STRINGS

In previous chapters, we've seen source-code examples using the **String** type, which is a fixed-length string type — essentially, it's an array of characters. In many cases, this data type is good enough to deal with textual information. However, there are situations that require more advanced text processing. Ada offers alternative approaches for these cases:

- *Bounded strings*: similar to fixed-length strings, bounded strings have a maximum length, which is set at its instantiation. However, bounded strings are not arrays of characters. At any time, they can contain a string of varied length — provided this length is below or equal to the maximum length.
- *Unbounded strings*: similar to bounded strings, unbounded strings can contain strings of varied length. However, in addition to that, they don't have a maximum length. In this sense, they are very flexible.

The following sections present an overview of the different string types and common operations for string types.

21.1 String operations

Operations on standard (fixed-length) strings are available in the `Ada.Strings.Fixed` package. As mentioned previously, standard strings are arrays of elements of **Character** type with a *fixed-length*. That's why this child package is called `Fixed`.

One of the simplest operations provided is counting the number of substrings available in a string (**Count**) and finding their corresponding indices (**Index**). Let's look at an example:

Listing 1: `show_find_substring.adb`

```
1  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  procedure Show_Find_Substring is
5
6    S   : String := "Hello" & 3 * "World";
7    P   : constant String := "World";
8    Idx : Natural;
9    Cnt : Natural;
10   begin
11     Cnt := Ada.Strings.Fixed.Count
12       (Source => S,
13        Pattern => P);
14
15     Put_Line ("String: " & S);
16     Put_Line ("Count for " & P & "'": "
17               & Natural'Image (Cnt));
```

(continues on next page)

(continued from previous page)

```

19  Idx := 0;
20  for I in 1 .. Cnt loop
21      Idx := Index
22          (Source => S,
23           Pattern => P,
24           From    => Idx + 1);
25
26      Put_Line ("Found instance of '" 
27                  & P & "' at position: " 
28                  & Natural'Image (Idx));
29  end loop;
30
31 end Show_Find_Substring;

```

Build output

```
show_find_substring.adb:6:04: warning: "S" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```

String: Hello World World World
Count for 'World': 3
Found instance of 'World' at position: 7
Found instance of 'World' at position: 13
Found instance of 'World' at position: 19

```

We initialize the string S using a multiplication. Writing "`Hello` & `3 * "World"` creates the string `Hello World World World`. We then call the function `Count` to get the number of instances of the word `World` in S. Next we call the function `Index` in a loop to find the index of each instance of `World` in S.

That example looked for instances of a specific substring. In the next example, we retrieve all the words in the string. We do this using `Find_Token` and specifying whitespaces as separators. For example:

Listing 2: `show_find_words.adb`

```

1  with Ada.Strings;      use Ada.Strings;
2  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3  with Ada.Strings.Maps;  use Ada.Strings.Maps;
4  with Ada.Text_Io;       use Ada.Text_Io;
5
6  procedure Show_Find_Words is
7
8      S   : String := "Hello" & 3 * " World";
9      F   : Positive;
10     L   : Natural;
11     I   : Natural := 1;
12
13     Whitespace : constant Character_Set :=
14         To_Set (' ');
15
16 begin
17     Put_Line ("String: " & S);
18     Put_Line ("String length: "
19                 & Integer'Image (S'Length));
20
21     while I in S'Range loop
22         Find_Token
23             (Source => S,
24              Set    => Whitespace,

```

(continues on next page)

(continued from previous page)

```

24      From    => I,
25      Test    => Outside,
26      First   => F,
27      Last    => L);
28
29      exit when L = 0;
30
31      Put_Line ("Found word instance at position "
32                  & Natural'Image (F)
33                  & ":" & S (F .. L) & "'");
34      -- & "-" & F'Img & "-" & L'Img
35
36      I := L + 1;
37  end loop;
38 end Show_Find_Words;

```

Build output

```
show_find_words.adb:8:04: warning: "S" is not modified, could be declared constant ↵
↪ [-gnatwk]
```

Runtime output

```

String: Hello World World World
String length: 23
Found word instance at position 1: 'Hello'
Found word instance at position 7: 'World'
Found word instance at position 13: 'World'
Found word instance at position 19: 'World'
```

We pass a set of characters to be used as delimitators to the procedure `Find_Token`. This set is a member of the `Character_Set` type from the `Ada.Strings.Maps` package. We call the `To_Set` function (from the same package) to initialize the set to `Whitespace` and then call `Find_Token` to loop over each valid index and find the starting index of each word. We pass `Outside` to the `Test` parameter of the `Find_Token` procedure to indicate that we're looking for indices that are outside the `Whitespace` set, i.e. actual words. The `First` and `Last` parameters of `Find_Token` are output parameters that indicate the valid range of the substring. We use this information to display the string (`S (F .. L)`).

The operations we've looked at so far read strings, but don't modify them. We next discuss operations that change the content of strings:

Operation	Description
Insert	Insert substring in a string
Overwrite	Overwrite a string with a substring
Delete	Delete a substring
Trim	Remove whitespaces from a string

All these operations are available both as functions or procedures. Functions create a new string but procedures perform the operations in place. The procedure will raise an exception if the constraints of the string are not satisfied. For example, if we have a string `S` containing 10 characters, inserting a string with two characters (e.g. `"!!"`) into it produces a string containing 12 characters. Since it has a fixed length, we can't increase its size. One possible solution in this case is to specify that truncation should be applied while inserting the substring. This keeps the length of `S` fixed. Let's see an example that makes use of both function and procedure versions of `Insert`, `Overwrite`, and `Delete`:

Listing 3: show_adapted_strings.adb

```

1  with Ada.Strings;      use Ada.Strings;
2  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3  with Ada.Text_IO;       use Ada.Text_IO;
4
5  procedure Show_Adapted_Strings is
6
7    S   : String := "Hello World";
8    P   : constant String := "World";
9    N   : constant String := "Beautiful";
10
11  procedure Display_Adapted_String
12    (Source   : String;
13     Before   : Positive;
14     New_Item : String;
15     Pattern  : String)
16  is
17    S_Ins_In : String := Source;
18    S_Ovr_In : String := Source;
19    S_Del_In : String := Source;
20
21    S_Ins : String :=
22      Insert (Source,
23              Before,
24              New_Item & " ");
25    S_Ovr : String :=
26      Overwrite (Source,
27                  Before,
28                  New_Item);
29    S_Del : String :=
30      Trim (Delete (Source,
31                     Before,
32                     Before + Pattern'Length - 1),
33            Ada.Strings.Right);
34
35  begin
36    Insert (S_Ins_In,
37            Before,
38            New_Item,
39            Right);
40
41    Overwrite (S_Ovr_In,
42               Before,
43               New_Item,
44               Right);
45
46    Delete (S_Del_In,
47            Before,
48            Before + Pattern'Length - 1);
49
50    Put_Line ("Original:   "
51              & Source & "'");
52
53    Put_Line ("Insert:     "
54              & S_Ins & "'");
55    Put_Line ("Overwrite:   "
56              & S_Ovr & "'");
57    Put_Line ("Delete:     "
58              & S_Del & "'");
59
60    Put_Line ("Insert (in-place): "
61              & S_Ins_In & "'");

```

(continues on next page)

(continued from previous page)

```

61      Put_Line ("Overwrite (in-place): ''"
62          & S_Ovr_In & "'");
63      Put_Line ("Delete    (in-place): ''"
64          & S_Del_In & "'");
65  end Display_Adapted_String;
66
67  Idx : Natural;
68 begin
69  Idx := Index
70      (Source => S,
71       Pattern => P);
72
73  if Idx > 0 then
74      Display_Adapted_String (S, Idx, N, P);
75  end if;
76 end Show_Adapted.Strings;

```

Build output

```

show_adapted_strings.adb:7:04: warning: "S" is not modified, could be declared
  ↵constant [-gnatwk]
show_adapted_strings.adb:21:07: warning: "S_Ins" is not modified, could be
  ↵declared constant [-gnatwk]
show_adapted_strings.adb:25:07: warning: "S_Ovr" is not modified, could be
  ↵declared constant [-gnatwk]
show_adapted_strings.adb:29:07: warning: "S_Del" is not modified, could be
  ↵declared constant [-gnatwk]

```

Runtime output

```

Original: 'Hello World'
Insert:  'Hello Beautiful World'
Overwrite: 'Hello Beautiful'
Delete:  'Hello'
Insert  (in-place): 'Hello Beaut'
Overwrite (in-place): 'Hello Beaut'
Delete   (in-place): 'Hello '

```

In this example, we look for the index of the substring `World` and perform operations on this substring within the outer string. The procedure `Display_Adapted_String` uses both versions of the operations. For the procedural version of `Insert` and `Overwrite`, we apply truncation to the right side of the string (Right). For the `Delete` procedure, we specify the range of the substring, which is replaced by whitespaces. For the function version of `Delete`, we also call `Trim` which trims the trailing whitespace.

21.2 Limitation of fixed-length strings

Using fixed-length strings is usually good enough for strings that are initialized when they are declared. However, as seen in the previous section, procedural operations on strings cause difficulties when done on fixed-length strings because fixed-length strings are arrays of characters. The following example shows how cumbersome the initialization of fixed-length strings can be when it's not performed in the declaration:

Listing 4: `show_char_array.adb`

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2

```

(continues on next page)

(continued from previous page)

```

3  procedure Show_Char_Array is
4    S : String (1 .. 15);
5    -- Strings are arrays of Character
6  begin
7    S := "Hello      ";
8    -- Alternatively:
9    --
10   -- #1:
11   --   S (1 .. 5)      := "Hello";
12   --   S (6 .. S'Last) := (others => ' ');
13   --
14   -- #2:
15   --   S := ('H', 'e', 'l', 'l', 'o',
16   --           others => ' ');
17
18   Put_Line ("String: " & S);
19   Put_Line ("String Length: "
20             & Integer'Image (S'Length));
21 end Show_Char_Array;

```

Runtime output

```

String: Hello
String Length: 15

```

In this case, we can't simply write `S := "Hello"` because the resulting array of characters for the `Hello` constant has a different length than the `S` string. Therefore, we need to include trailing whitespaces to match the length of `S`. As shown in the example, we could use an exact range for the initialization (`S (1 .. 5)`) or use an explicit array of individual characters.

When strings are initialized or manipulated at run-time, it's usually better to use bounded or unbounded strings. An important feature of these types is that they aren't arrays, so the difficulties presented above don't apply. Let's start with bounded strings.

21.3 Bounded strings

Bounded strings are defined in the `Ada.Strings.Bounded.Generic_Bounded_Length` package. Because this is a generic package, you need to instantiate it and set the maximum length of the bounded string. You can then declare bounded strings of the `Bounded_String` type.

Both bounded and fixed-length strings have a maximum length that they can hold. However, bounded strings are not arrays, so initializing them at run-time is much easier. For example:

Listing 5: show_bounded_string.adb

```

1  with Ada.Strings;          use Ada.Strings;
2  with Ada.Strings.Bounded;   use Ada.Strings.Bounded;
3  with Ada.Text_IO;          use Ada.Text_IO;
4
5  procedure Show_Bounded_String is
6    package B_Str is new
7      Ada.Strings.Bounded.Generic_Bounded_Length (Max => 15);
8    use B_Str;
9
10   S1, S2 : Bounded_String;

```

(continues on next page)

(continued from previous page)

```

11
12 procedure Display_String_Info (S : Bounded_String) is
13 begin
14   Put_Line ("String: " & To_String (S));
15   Put_Line ("String Length: "
16             & Integer'Image (Length (S)));
17   -- String:
18   --      S'Length => ok
19   -- Bounded_String:
20   --      S'Length => compilation error:
21   --                  bounded strings are
22   --                  not arrays!
23
24   Put_Line ("Max. Length: "
25             & Integer'Image (Max_Length));
26 end Display_String_Info;
27 begin
28   S1 := To_Bounded_String ("Hello");
29   Display_String_Info (S1);
30
31   S2 := To_Bounded_String ("Hello World");
32   Display_String_Info (S2);
33
34   S1 := To_Bounded_String
35   ("Something longer to say here...",
36    Right);
37   Display_String_Info (S1);
38 end Show_Bounded_String;

```

Runtime output

```

String: Hello
String Length: 5
Max. Length: 15
String: Hello World
String Length: 11
Max. Length: 15
String: Something longe
String Length: 15
Max. Length: 15

```

By using bounded strings, we can easily assign to S1 and S2 multiple times during execution. We use the To_Bounded_String and To_String functions to convert, in the respective direction, between fixed-length and bounded strings. A call to To_Bounded_String raises an exception if the length of the input string is greater than the maximum capacity of the bounded string. To avoid this, we can use the truncation parameter (Right in our example).

Bounded strings are not arrays, so we can't use the 'Length attribute as we did for fixed-length strings. Instead, we call the Length function, which returns the length of the bounded string. The Max_Length constant represents the maximum length of the bounded string that we set when we instantiated the package.

After initializing a bounded string, we can manipulate it. For example, we can append a string to a bounded string using Append or concatenate bounded strings using the & operator. Like so:

Listing 6: show_bounded_string_op.adb

```

1 with Ada.Strings;          use Ada.Strings;
2 with Ada.Strings.Bounded;
3 with Ada.Text_IO;          use Ada.Text_IO;

```

(continues on next page)

(continued from previous page)

```

4
5 procedure Show_Bounded_String_Op is
6   package B_Str is new
7     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 30);
8   use B_Str;
9
10  S1, S2 : Bounded_String;
11 begin
12   S1 := To_Bounded_String ("Hello");
13   -- Alternatively:
14   --
15   -- A := Null_Bounded_String & "Hello";
16
17   Append (S1, " World");
18   -- Alternatively: Append (A, " World", Right);
19
20   Put_Line ("String: " & To_String (S1));
21
22   S2 := To_Bounded_String ("Hello!");
23   S1 := S1 & " " & S2;
24   Put_Line ("String: " & To_String (S1));
25 end Show_Bounded_String_Op;

```

Runtime output

```

String: Hello World
String: Hello World Hello!

```

We can initialize a bounded string with an empty string using the `Null_Bounded_String` constant. Also, we can use the `Append` procedure and specify the truncation mode like we do with the `To_Bounded_String` function.

21.4 Unbounded strings

Unbounded strings are defined in the `Ada.Strings.Unbounded` package. This is *not* a generic package, so we don't need to instantiate it before using the `Unbounded_String` type. As you may recall from the previous section, bounded strings require a package instantiation.

Unbounded strings are similar to bounded strings. The main difference is that they can hold strings of any size and adjust according to the input string: if we assign, e.g., a 10-character string to an unbounded string and later assign a 50-character string, internal operations in the container ensure that memory is allocated to store the new string. In most cases, developers don't need to worry about these operations. Also, no truncation is necessary.

Initialization of unbounded strings is very similar to bounded strings. Let's look at an example:

Listing 7: show_unbounded_string.adb

```

1 with Ada.Strings;           use Ada.Strings;
2 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
3 with Ada.Text_Io;           use Ada.Text_Io;
4
5 procedure Show_Unbounded_String is
6   S1, S2 : Unbounded_String;
7

```

(continues on next page)

(continued from previous page)

```

8   procedure Display_String_Info (S : Unbounded_String) is
9   begin
10    Put_Line ("String: " & To_String (S));
11    Put_Line ("String Length: "
12              & Integer'Image (Length (S)));
13   end Display_String_Info;
14 begin
15  S1 := To_Unbounded_String ("Hello");
16  -- Alternatively:
17  --
18  -- A := Null_Unbounded_String & "Hello";
19
20  Display_String_Info (S1);
21
22  S2 := To_Unbounded_String ("Hello World");
23  Display_String_Info (S2);
24
25  S1 := To_Unbounded_String ("Something longer to say here...");
26  Display_String_Info (S1);
27 end Show_Unbounded_String;

```

Runtime output

```

String: Hello
String Length: 5
String: Hello World
String Length: 11
String: Something longer to say here...
String Length: 31

```

Like bounded strings, we can assign to S1 and S2 multiple times during execution and use the To_Unbounded_String and To_String functions to convert back-and-forth between fixed-length strings and unbounded strings. However, in this case, truncation is not needed.

And, just like for bounded strings, you can use the Append procedure and the & operator for unbounded strings. For example:

Listing 8: show_unbounded_string_op.adb

```

1  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2  with Ada.Text_Io;           use Ada.Text_Io;
3
4  procedure Show_Unbounded_String_Op is
5    S1, S2 : Unbounded_String := Null_Unbounded_String;
6  begin
7    S1 := S1 & "Hello";
8    S2 := S2 & "Hello!";
9
10   Append (S1, " World");
11   Put_Line ("String: " & To_String (S1));
12
13   S1 := S1 & " " & S2;
14   Put_Line ("String: " & To_String (S1));
15 end Show_Unbounded_String_Op;

```

Runtime output

```

String: Hello World
String: Hello World Hello!

```

CHAPTER
TWENTYTWO

STANDARD LIBRARY: FILES AND STREAMS

Ada provides different approaches for file input/output (I/O):

- *Text I/O*, which supports file I/O in text format, including the display of information on the console.
- *Sequential I/O*, which supports file I/O in binary format written in a sequential fashion for a specific data type.
- *Direct I/O*, which supports file I/O in binary format for a specific data type, but also supporting access to any position of a file.
- *Stream I/O*, which supports I/O of information for multiple data types, including objects of unbounded types, using files in binary format.

This table presents a summary of the features we've just seen:

File I/O option	Format	Random access	Data types
Text I/O	text		string type
Sequential I/O	binary		single type
Direct I/O	binary	✓	single type
Stream I/O	binary	✓	multiple types

In the following sections, we discuss details about these I/O approaches.

22.1 Text I/O

In most parts of this course, we used the `Put_Line` procedure to display information on the console. However, this procedure also accepts a **File_Type** parameter. For example, you can select between standard output and standard error by setting this parameter explicitly:

Listing 1: `show_std_text_out.adb`

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Show_Std_Text_Out is
4 begin
5   Put_Line (Standard_Output, "Hello World #1");
6   Put_Line (Standard_Error, "Hello World #2");
7 end Show_Std_Text_Out;
```

Runtime output

```
Hello World #1
Hello World #2
```

You can also use this parameter to write information to any text file. To create a new file for writing, use the Create procedure, which initializes a **File_Type** element that you can later pass to Put_Line (instead of, e.g., Standard_Output). After you finish writing information, you can close the file by calling the Close procedure.

You use a similar method to read information from a text file. However, when opening the file, you must specify that it's an input file (In_File) instead of an output file. Also, instead of calling the Put_Line procedure, you call the Get_Line function to read information from the file.

Let's see an example that writes information into a new text file and then reads it back from the same file:

Listing 2: show_simple_text_file_io.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Simple_Text_File_IO is
4      F       : File_Type;
5      File_Name : constant String := "simple.txt";
6  begin
7      Create (F, Out_File, File_Name);
8      Put_Line (F, "Hello World #1");
9      Put_Line (F, "Hello World #2");
10     Put_Line (F, "Hello World #3");
11     Close (F);
12
13     Open (F, In_File, File_Name);
14     while not End_Of_File (F) loop
15         Put_Line (Get_Line (F));
16     end loop;
17     Close (F);
18 end Show_Simple_Text_File_IO;
```

Runtime output

```
Hello World #1
Hello World #2
Hello World #3
```

In addition to the Create and Close procedures, the standard library also includes a Reset procedure, which, as the name implies, resets (erases) all the information from the file. For example:

Listing 3: show_text_file_reset.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Text_File_Reset is
4      F       : File_Type;
5      File_Name : constant String := "simple.txt";
6  begin
7      Create (F, Out_File, File_Name);
8      Put_Line (F, "Hello World #1");
9      Reset (F);
10     Put_Line (F, "Hello World #2");
11     Close (F);
12
13     Open (F, In_File, File_Name);
14     while not End_Of_File (F) loop
15         Put_Line (Get_Line (F));
16     end loop;
```

(continues on next page)

(continued from previous page)

```

17   Close (F);
18 end Show_Text_File_Reset;

```

Runtime output

```
Hello World #2
```

By running this program, we notice that, although we've written the first string ("Hello World #1") to the file, it has been erased because of the call to Reset.

In addition to opening a file for reading or writing, you can also open an existing file and append to it. Do this by calling the Open procedure with the Append_File option.

When calling the Open procedure, an exception is raised if the specified file isn't found. Therefore, you should handle exceptions in that context. The following example deletes a file and then tries to open the same file for reading:

Listing 4: show_text_file_input_except.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Text_File_Input_Except is
4    F        : File_Type;
5    File_Name : constant String := "simple.txt";
6  begin
7    -- Open output file and delete it
8    Create (F, Out_File, File_Name);
9    Delete (F);
10
11   -- Try to open deleted file
12   Open (F, In_File, File_Name);
13   Close (F);
14 exception
15   when Name_Error =>
16     Put_Line ("File does not exist");
17   when others =>
18     Put_Line ("Error while processing input file");
19 end Show_Text_File_Input_Except;

```

Runtime output

```
File does not exist
```

In this example, we create the file by calling Create and then delete it by calling Delete. After the call to Delete, we can no longer use the **File_Type** element. After deleting the file, we try to open the non-existent file, which raises a **Name_Error** exception.

22.2 Sequential I/O

The previous section presented details about text file I/O. Here, we discuss doing file I/O in binary format. The first package we'll explore is the **Ada.Sequential_Io** package. Because this package is a generic package, you need to instantiate it for the data type you want to use for file I/O. Once you've done that, you can use the same procedures we've seen in the previous section: Create, Open, Close, Reset and Delete. However, instead of calling the Get_Line and Put_Line procedures, you'd call the Read and Write procedures.

In the following example, we instantiate the **Ada.Sequential_Io** package for floating-point types:

Listing 5: show_seq_float_io.adb

```

1  with Ada.Text_IO;
2  with Ada.Sequential_IO;
3
4  procedure Show_Seq_Float_IO is
5    package Float_IO is
6      new Ada.Sequential_IO (Float);
7    use Float_IO;
8
9    F       : Float_IO.File_Type;
10   File_Name : constant String := "float_file.bin";
11
12 begin
13   Create (F, Out_File, File_Name);
14   Write (F, 1.5);
15   Write (F, 2.4);
16   Write (F, 6.7);
17   Close (F);
18
19 declare
20   Value : Float;
21 begin
22   Open (F, In_File, File_Name);
23   while not End_Of_File (F) loop
24     Read (F, Value);
25     Ada.Text_IO.Put_Line (Float'Image (Value));
26   end loop;
27   Close (F);
28 end;
end Show_Seq_Float_IO;

```

Runtime output

```

1.50000E+00
2.40000E+00
6.70000E+00

```

We use the same approach to read and write complex information. The following example uses a record that includes a Boolean and a floating-point value:

Listing 6: show_seq_rec_io.adb

```

1  with Ada.Text_IO;
2  with Ada.Sequential_IO;
3
4  procedure Show_Seq_Rec_IO is
5    type Num_Info is record
6      Valid : Boolean := False;
7      Value : Float;
8    end record;
9
10 procedure Put_Line (N : Num_Info) is
11 begin
12   if N.Valid then
13     Ada.Text_IO.Put_Line ("(ok,      "
14                           & Float'Image (N.Value) & ")");
15   else
16     Ada.Text_IO.Put_Line ("(not ok, -----)");
17   end if;
18 end Put_Line;
19

```

(continues on next page)

(continued from previous page)

```

20 package Num_Info_IO is new Ada.Sequential_IO (Num_Info);
21 use Num_Info_IO;
22
23   F      : Num_Info_IO.File_Type;
24   File_Name : constant String := "float_file.bin";
25 begin
26   Create (F, Out_File, File_Name);
27   Write (F, (True, 1.5));
28   Write (F, (False, 2.4));
29   Write (F, (True, 6.7));
30   Close (F);
31
32 declare
33   Value : Num_Info;
34 begin
35   Open (F, In_File, File_Name);
36   while not End_Of_File (F) loop
37     Read (F, Value);
38     Put_Line (Value);
39   end loop;
40   Close (F);
41 end;
42 end Show_Seq_Rec_I0;

```

Runtime output

```

(ok,      1.50000E+00)
(not ok,  -----
(ok,      6.70000E+00)

```

As the example shows, we can use the same approach we used for floating-point types to perform file I/O for this record. Once we instantiate the Ada.Sequential_IO package for the record type, file I/O operations are performed the same way.

22.3 Direct I/O

Direct I/O is available in the Ada.Direct_IO package. This mechanism is similar to the sequential I/O approach just presented, but allows us to access any position in the file. The package instantiation and most operations are very similar to sequential I/O. To rewrite the Show_Seq_Float_I0 application presented in the previous section to use the Ada.Direct_IO package, we just need to replace the instances of the Ada.Sequential_IO package by the Ada.Direct_IO package. This is the new source code:

Listing 7: show_dir_float_io.adb

```

1 with Ada.Text_IO;
2 with Ada.Direct_IO;
3
4 procedure Show_Dir_Float_I0 is
5   package Float_I0 is new Ada.Direct_IO (Float);
6   use Float_I0;
7
8   F      : Float_I0.File_Type;
9   File_Name : constant String := "float_file.bin";
10 begin
11   Create (F, Out_File, File_Name);
12   Write (F, 1.5);

```

(continues on next page)

(continued from previous page)

```

13  Write (F,  2.4);
14  Write (F,  6.7);
15  Close (F);

16
17  declare
18      Value : Float;
19  begin
20      Open (F, In_File, File_Name);
21      while not End_Of_File (F) loop
22          Read (F, Value);
23          Ada.Text_IO.Put_Line (Float'Image (Value));
24      end loop;
25      Close (F);
26  end;
27 end Show_Dir_Float_IO;

```

Runtime output

```

1.50000E+00
2.40000E+00
6.70000E+00

```

Unlike sequential I/O, direct I/O allows you to access any position in the file. However, it doesn't offer an option to append information to a file. Instead, it provides an `Inout_File` mode allowing reading and writing to a file via the same `File_Type` element.

To access any position in the file, call the `Set_Index` procedure to set the new position / index. You can use the `Index` function to retrieve the current index. Let's see an example:

Listing 8: show_dir_float_in_out_file.adb

```

1  with Ada.Text_IO;
2  with Ada.Direct_IO;
3
4  procedure Show_Dir_Float_In_Out_File is
5      package Float_IO is new Ada.Direct_IO (Float);
6      use Float_IO;
7
8      F        : Float_IO.File_Type;
9      File_Name : constant String := "float_file.bin";
10     begin
11         -- Open file for input / output
12         Create (F, Inout_File, File_Name);
13         Write (F, 1.5);
14         Write (F, 2.4);
15         Write (F, 6.7);
16
17         -- Set index to previous position and overwrite value
18         Set_Index (F, Index (F) - 1);
19         Write (F, 7.7);
20
21     declare
22         Value : Float;
23     begin
24         -- Set index to start of file
25         Set_Index (F, 1);
26
27         while not End_Of_File (F) loop
28             Read (F, Value);
29             Ada.Text_IO.Put_Line (Float'Image (Value));
30         end loop;

```

(continues on next page)

(continued from previous page)

```

31     Close (F);
32   end;
33 end Show_Dir_Float_In_Out_File;
```

Runtime output

```

1.50000E+00
2.40000E+00
7.70000E+00
```

By running this example, we see that the file contains 7.7, rather than the previous 6.7 that we wrote. We overwrote the value by changing the index to the previous position before doing another write.

In this example we used the Inout_File mode. Using that mode, we just changed the index back to the initial position before reading from the file (Set_Index (F, 1)) instead of closing the file and reopening it for reading.

22.4 Stream I/O

All the previous approaches for file I/O in binary format (sequential and direct I/O) are specific for a single data type (the one we instantiate them with). You can use these approaches to write objects of a single data type that may be an array or record (potentially with many fields), but if you need to create and process files that include different data types, or any objects of an unbounded type, these approaches are not sufficient. Instead, you should use stream I/O.

Stream I/O shares some similarities with the previous approaches. We still use the Create, Open and Close procedures. However, instead of accessing the file directly via a **File_Type** element, you use a Stream_Access element. To read and write information, you use the **'Read** or **'Write** attributes of the data types you're reading or writing.

Let's look at a version of the Show_Dir_Float_I0 procedure from the previous section that makes use of stream I/O instead of direct I/O:

Listing 9: show_float_stream.adb

```

1  with Ada.Text_IO;
2  with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;
3
4  procedure Show_Float_Stream is
5    F        : File_Type;
6    S        : Stream_Access;
7    File_Name : constant String := "float_file.bin";
8  begin
9    Create (F, Out_File, File_Name);
10   S := Stream (F);
11
12   Float'Write (S, 1.5);
13   Float'Write (S, 2.4);
14   Float'Write (S, 6.7);
15
16   Close (F);
17
18   declare
19     Value : Float;
20   begin
21     Open (F, In_File, File_Name);
```

(continues on next page)

(continued from previous page)

```

22      S := Stream (F);
23
24      while not End_Of_File (F) loop
25          Float'Read (S, Value);
26          Ada.Text_IO.Put_Line (Float'Image (Value));
27      end loop;
28      Close (F);
29  end;
end Show_Float_Stream;

```

Runtime output

```

1.50000E+00
2.40000E+00
6.70000E+00

```

After the call to Create, we retrieve the corresponding Stream_Access element by calling the Stream function. We then use this stream to write information to the file via the '[Write](#)' attribute of the **Float** type. After closing the file and reopening it for reading, we again retrieve the corresponding Stream_Access element and processed to read information from the file via the '[Read](#)' attribute of the **Float** type.

You can use streams to create and process files containing different data types within the same file. You can also read and write unbounded data types such as strings. However, when using unbounded data types you must call the '[Input](#)' and '[Output](#)' attributes of the unbounded data type: these attributes write information about bounds or discriminants in addition to the object's actual data.

The following example shows file I/O that mixes both strings of different lengths and floating-point values:

Listing 10: show_string_stream.adb

```

1  with Ada.Text_IO;
2  with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;
3
4  procedure Show_String_Stream is
5      F        : File_Type;
6      S        : Stream_Access;
7      File_Name : constant String := "float_file.bin";
8
9      procedure Output (S : Stream_Access;
10                     FV : Float;
11                     SV : String) is
12
13      begin
14          String'Output (S, SV);
15          Float'Output (S, FV);
16      end Output;
17
18      procedure Input_Display (S : Stream_Access) is
19          SV : String := String'Input (S);
20          FV : Float   := Float'Input (S);
21
22      begin
23          Ada.Text_IO.Put_Line (Float'Image (FV)
24                               & " --- " & SV);
25      end Input_Display;
26
27  begin
28      Create (F, Out_File, File_Name);
29      S := Stream (F);

```

(continues on next page)

(continued from previous page)

```

29   Output (S, 1.5, "Hi!!");
30   Output (S, 2.4, "Hello world!");
31   Output (S, 6.7, "Something longer here...");

32   Close (F);

33   Open (F, In_File, File_Name);
34   S := Stream (F);

35   while not End_Of_File (F) loop
36     Input_Display (S);
37   end loop;
38   Close (F);

39 end Show_String_Stream;

```

Build output

```

show_string_stream.adb:18:07: warning: "SV" is not modified, could be declared constant [-gnatwk]
show_string_stream.adb:19:07: warning: "FV" is not modified, could be declared constant [-gnatwk]

```

Runtime output

```

1.50000E+00 --- Hi!!
2.40000E+00 --- Hello world!
6.70000E+00 --- Something longer here...

```

When you use Stream I/O, no information is written into the file indicating the type of the data that you wrote. If a file contains data from different types, you must reference types in the same order when reading a file as when you wrote it. If not, the information you get will be corrupted. Unfortunately, strong data typing doesn't help you in this case. Writing simple procedures for file I/O (as in the example above) may help ensuring that the file format is consistent.

Like direct I/O, stream I/O supports also allows you to access any location in the file. However, when doing so, you need to be extremely careful that the position of the new index is consistent with the data types you're expecting.

CHAPTER
TWENTYTHREE

STANDARD LIBRARY: NUMERICS

The standard library provides support for common numeric operations on floating-point types as well as on complex types and matrices. In the sections below, we present a brief introduction to these numeric operations.

23.1 Elementary Functions

The `Ada.Numerics.Elementary_Functions` package provides common operations for floating-point types, such as square root, logarithm, and the trigonometric functions (e.g., `sin`, `cos`). For example:

Listing 1: `show_elem_math.adb`

```
1  with Ada.Text_Io;  use Ada.Text_Io;
2  with Ada.Numerics; use Ada.Numerics;
3
4  with Ada.Numerics.Elementary_Functions;
5  use Ada.Numerics.Elementary_Functions;
6
7  procedure Show_Elem_Math is
8      X : Float;
9  begin
10     X := 2.0;
11     Put_Line ("Square root of "
12               & Float'Image (X)
13               & " is "
14               & Float'Image (Sqrt (X)));
15
16     X := e;
17     Put_Line ("Natural log of "
18               & Float'Image (X)
19               & " is "
20               & Float'Image (Log (X)));
21
22     X := 10.0 ** 6.0;
23     Put_Line ("Log_10      of "
24               & Float'Image (X)
25               & " is "
26               & Float'Image (Log (X, 10.0)));
27
28     X := 2.0 ** 8.0;
29     Put_Line ("Log_2      of "
30               & Float'Image (X)
31               & " is "
32               & Float'Image (Log (X, 2.0)));
33
```

(continues on next page)

(continued from previous page)

```

34  X := Pi;
35  Put_Line ("Cos      of "
36    & Float'Image (X)
37    & " is "
38    & Float'Image (Cos (X)));
39
40  X := -1.0;
41  Put_Line ("Arccos     of "
42    & Float'Image (X)
43    & " is "
44    & Float'Image (Arccos (X)));
45 end Show_Elem_Math;

```

Runtime output

```

Square root of 2.00000E+00 is 1.41421E+00
Natural log of 2.71828E+00 is 1.00000E+00
Log_10      of 1.00000E+06 is 6.00000E+00
Log_2       of 2.56000E+02 is 8.00000E+00
Cos         of 3.14159E+00 is -1.00000E+00
Arccos      of -1.00000E+00 is 3.14159E+00

```

Here we use the standard e and Pi constants from the Ada.Numerics package.

The Ada.Numerics.Elementary_Functions package provides operations for the **Float** type. Similar packages are available for **Long_Float** and **Long_Long_Float** types. For example, the Ada.Numerics.Long_Elementary_Functions package offers the same set of operations for the **Long_Float** type. In addition, the Ada.Numerics.**Generic_Elementary_Functions** package is a generic version of the package that you can instantiate for custom floating-point types. In fact, the Elementary_Functions package can be defined as follows:

```
package Elementary_Functions is new
  Ada.Numerics.Generic_Elementary_Functions (Float);
```

23.2 Random Number Generation

The Ada.Numerics.Float_Random package provides a simple random number generator for the range between 0.0 and 1.0. To use it, declare a generator G, which you pass to Random. For example:

Listing 2: show_float_random_num.adb

```

1  with Ada.Text_Io;  use Ada.Text_Io;
2  with Ada.Numerics.Float_Random; use Ada.Numerics.Float_Random;
3
4  procedure Show_Float_Random_Num is
5    G : Generator;
6    X : Uniformly_Distributed;
7  begin
8    Reset (G);
9
10   Put_Line ("Some random numbers between "
11     & Float'Image (Uniformly_Distributed'First)
12     & " and "
13     & Float'Image (Uniformly_Distributed'Last)
14     & ":");


```

(continues on next page)

(continued from previous page)

```

15  for I in 1 .. 15 loop
16    X := Random (G);
17    Put_Line (Float'Image (X));
18  end loop;
19 end Show_Float_Random_Num;

```

Runtime output

Some random numbers between 0.00000E+00 and 1.00000E+00:

```

8.40236E-01
2.10435E-01
1.06138E-01
1.31775E-01
1.59622E-01
6.76792E-01
2.93800E-01
1.80331E-02
9.33314E-01
8.38966E-01
2.13388E-01
8.33035E-01
7.81441E-01
4.49502E-01
7.85728E-01

```

The standard library also includes a random number generator for discrete numbers, which is part of the `Ada.Numerics.Discrete_Random` package. Since it's a generic package, you have to instantiate it for the desired discrete type. This allows you to specify a range for the generator. In the following example, we create an application that displays random integers between 1 and 10:

Listing 3: `show_discrete_random_num.adb`

```

1  with Ada.Text_Io;  use Ada.Text_Io;
2  with Ada.Numerics.Discrete_Random;
3
4  procedure Show_Discrete_Random_Num is
5
6    subtype Random_Range is Integer range 1 .. 10;
7
8    package R is new
9      Ada.Numerics.Discrete_Random (Random_Range);
10   use R;
11
12   G : Generator;
13   X : Random_Range;
14 begin
15   Reset (G);
16
17   Put_Line ("Some random numbers between "
18             & Integer'Image (Random_Range'First)
19             & " and "
20             & Integer'Image (Random_Range'Last)
21             & ":" );
22
23   for I in 1 .. 15 loop
24     X := Random (G);
25     Put_Line (Integer'Image (X));
26   end loop;
27 end Show_Discrete_Random_Num;

```

Runtime output

```
Some random numbers between 1 and 10:  
10  
10  
7  
8  
1  
1  
3  
2  
5  
5  
6  
1  
9  
8  
7
```

Here, package R is instantiated with the Random_Range type, which has a constrained range between 1 and 10. This allows us to control the range used for the random numbers. We could easily modify the application to display random integers between 0 and 20 by changing the specification of the Random_Range type. We can also use floating-point or fixed-point types.

23.3 Complex Types

The Ada.Numerics.Complex_Types package provides support for complex number types and the Ada.Numerics.Complex_Elementary_Functions package provides support for common operations on complex number types, similar to the Ada.Numerics.Elementary_Functions package. Finally, you can use the Ada.Text_Io.Complex_Io package to perform I/O operations on complex numbers. In the following example, we declare variables of the Complex type and initialize them using an aggregate:

Listing 4: show_elem_math.adb

```
1  with Ada.Text_Io;  use Ada.Text_Io;  
2  with Ada.Numerics; use Ada.Numerics;  
3  
4  with Ada.Numerics.Complex_Types;  
5  use Ada.Numerics.Complex_Types;  
6  
7  with Ada.Numerics.Complex_Elementary_Functions;  
8  use Ada.Numerics.Complex_Elementary_Functions;  
9  
10 with Ada.Text_Io.Complex_Io;  
11  
12 procedure Show_Elem_Math is  
13  
14  package C_IO is new  
15    Ada.Text_Io.Complex_Io (Complex_Types);  
16  use C_IO;  
17  
18  X, Y : Complex;  
19  R, Th : Float;  
20 begin  
21  X := (2.0, -1.0);  
22  Y := (3.0, 4.0);
```

(continues on next page)

(continued from previous page)

```

24 Put (X);
25 Put (" * ");
26 Put (Y);
27 Put (" is ");
28 Put (X * Y);
29 New_Line;
30 New_Line;

31
32 R := 3.0;
33 Th := Pi / 2.0;
34 X := Compose_From_Polar (R, Th);
-- Alternatively:
-- X := R * Exp ((0.0, Th));
-- X := R * e ** Complex'(0.0, Th);

35 Put ("Polar form:      "
36   & Float'Image (R) & " * e**(i * "
37   & Float'Image (Th) & ")");
38 New_Line;

39 Put ("Modulus      of ");
40 Put (X);
41 Put (" is ");
42 Put (Float'Image (abs (X)));
43 New_Line;

44 Put ("Argument      of ");
45 Put (X);
46 Put (" is ");
47 Put (Float'Image (Argument (X)));
48 New_Line;
49 New_Line;

50 Put ("Sqrt      of ");
51 Put (X);
52 Put (" is ");
53 Put (Sqrt (X));
54 New_Line;
55 New_Line;

56
57 end Show_Elem_Math;

```

Runtime output

```

( 2.00000E+00, -1.00000E+00) * ( 3.00000E+00, 4.00000E+00) is ( 1.00000E+01, 5.
00000E+00)

Polar form:      3.00000E+00 * e**(i * 1.57080E+00)
Modulus      of (-1.31134E-07, 3.00000E+00) is 3.00000E+00
Argument      of (-1.31134E-07, 3.00000E+00) is 1.57080E+00

Sqrt      of (-1.31134E-07, 3.00000E+00) is ( 1.22474E+00, 1.22474E+00)

```

As we can see from this example, all the common operators, such as `*` and `+`, are available for complex types. You also have typical operations on complex numbers, such as `Argument` and `Exp`. In addition to initializing complex numbers in the cartesian form using aggregates, you can do so from the polar form by calling the `Compose_From_Polar` function.

The `Ada.Numerics.Complex_Types` and `Ada.Numerics.Complex_Elementary_Functions` packages provide operations for the `Float` type. Similar packages are available for `Long_Float` and `Long_Long_Float` types. In addition, the `Ada.Numerics.Generic_Complex_Types` and `Ada.Numerics.Generic_Complex_Elementary_Functions` packages are generic versions that you can instantiate for custom or pre-defined floating-

point types. For example:

```
with Ada.Numerics.Generic_Complex_Types;
with Ada.Numerics.Generic_Complex_Elementary_Functions;
with Ada.Text_Io.Complex_Io;

procedure Show_Elem_Math is

    package Complex_Types is new
        Ada.Numerics.Generic_Complex_Types (Float);
    use Complex_Types;

    package Elementary_Functions is new
        Ada.Numerics.Generic_Complex_Elementary_Functions
        (Complex_Types);
    use Elementary_Functions;

    package C_Io is new Ada.Text_Io.Complex_Io
        (Complex_Types);
    use C_Io;

    X, Y : Complex;
    R, Th : Float;
```

23.4 Vector and Matrix Manipulation

The Ada.Numerics.Real_Arrays package provides support for vectors and matrices. It includes common matrix operations such as inverse, determinant, eigenvalues in addition to simpler operators such as matrix addition and multiplication. You can declare vectors and matrices using the Real_Vector and Real_Matrix types, respectively.

The following example uses some of the operations from the Ada.Numerics.Real_Arrays package:

Listing 5: show_matrix.adb

```
1  with Ada.Text_Io;  use Ada.Text_Io;
2
3  with Ada.Numerics.Real_Arrays;
4  use Ada.Numerics.Real_Arrays;
5
6  procedure Show_Matrix is
7
8      procedure Put_Vector (V : Real_Vector) is
9  begin
10         Put ("  ");
11         for I in V'Range loop
12             Put (Float'Image (V (I)) & " ");
13         end loop;
14         Put_Line ("");
15     end Put_Vector;
16
17     procedure Put_Matrix (M : Real_Matrix) is
18  begin
19      for I in M'Range (1) loop
20          Put ("  ");
21          for J in M'Range (2) loop
22              Put (Float'Image (M (I, J)) & " ");
23          end loop;
```

(continues on next page)

(continued from previous page)

```

24      Put_Line ("");
25  end loop;
26 end Put_Matrix;

27
28 V1      : Real_Vector := (1.0, 3.0);
29 V2      : Real_Vector := (75.0, 11.0);
30
31 M1      : Real_Matrix :=
32      ((1.0, 5.0, 1.0),
33      (2.0, 2.0, 1.0));
34 M2      : Real_Matrix :=
35      ((31.0, 11.0, 10.0),
36      (34.0, 16.0, 11.0),
37      (32.0, 12.0, 10.0),
38      (31.0, 13.0, 10.0));
39 M3      : Real_Matrix := ((1.0, 2.0),
40                           (2.0, 3.0));

41 begin
42   Put_Line ("V1");
43   Put_Vector (V1);
44   Put_Line ("V2");
45   Put_Vector (V2);
46   Put_Line ("V1 * V2 =");
47   Put_Line (""
48             & Float'Image (V1 * V2));
49   Put_Line ("V1 * V2 =");
50   Put_Matrix (V1 * V2);
51   New_Line;

52
53   Put_Line ("M1");
54   Put_Matrix (M1);
55   Put_Line ("M2");
56   Put_Matrix (M2);
57   Put_Line ("M2 * Transpose(M1) =");
58   Put_Matrix (M2 * Transpose (M1));
59   New_Line;

60
61   Put_Line ("M3");
62   Put_Matrix (M3);
63   Put_Line ("Inverse (M3) =");
64   Put_Matrix (Inverse (M3));
65   Put_Line ("abs Inverse (M3) =");
66   Put_Matrix (abs Inverse (M3));
67   Put_Line ("Determinant (M3) =");
68   Put_Line (""
69             & Float'Image (Determinant (M3)));
70   Put_Line ("Solve (M3, V1) =");
71   Put_Vector (Solve (M3, V1));
72   Put_Line ("Eigenvalues (M3) =");
73   Put_Vector (Eigenvalues (M3));
74   New_Line;
75 end Show_Matrix;

```

Build output

```

show_matrix.adb:28:04: warning: "V1" is not modified, could be declared constant [-gnatwk]
show_matrix.adb:29:04: warning: "V2" is not modified, could be declared constant [-gnatwk]
show_matrix.adb:31:04: warning: "M1" is not modified, could be declared constant [-gnatwk]

```

(continues on next page)

(continued from previous page)

```
show_matrix.adb:34:04: warning: "M2" is not modified, could be declared constant [-gnatwk]
show_matrix.adb:39:04: warning: "M3" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
V1
  ( 1.00000E+00  3.00000E+00 )
V2
  ( 7.50000E+01  1.10000E+01 )
V1 * V2 =
  1.08000E+02
V1 * V2 =
  ( 7.50000E+01  1.10000E+01 )
  ( 2.25000E+02  3.30000E+01 )

M1
  ( 1.00000E+00  5.00000E+00  1.00000E+00 )
  ( 2.00000E+00  2.00000E+00  1.00000E+00 )
M2
  ( 3.10000E+01  1.10000E+01  1.00000E+01 )
  ( 3.40000E+01  1.60000E+01  1.10000E+01 )
  ( 3.20000E+01  1.20000E+01  1.00000E+01 )
  ( 3.10000E+01  1.30000E+01  1.00000E+01 )
M2 * Transpose(M1) =
  ( 9.60000E+01  9.40000E+01 )
  ( 1.25000E+02  1.11000E+02 )
  ( 1.02000E+02  9.80000E+01 )
  ( 1.06000E+02  9.80000E+01 )

M3
  ( 1.00000E+00  2.00000E+00 )
  ( 2.00000E+00  3.00000E+00 )
Inverse (M3) =
  (-3.00000E+00  2.00000E+00 )
  ( 2.00000E+00 -1.00000E+00 )
abs Inverse (M3) =
  ( 3.00000E+00  2.00000E+00 )
  ( 2.00000E+00  1.00000E+00 )
Determinant (M3) =
  -1.00000E+00
Solve (M3, V1) =
  ( 3.00000E+00 -1.00000E+00 )
Eigenvalues (M3) =
  ( 4.23607E+00 -2.36068E-01 )
```

Matrix dimensions are automatically determined from the aggregate used for initialization when you don't specify them. You can, however, also use explicit ranges. For example:

```
M1      : Real_Matrix (1 .. 2, 1 .. 3) :=
  ((1.0, 5.0, 1.0),
   (2.0, 2.0, 1.0));
```

The Ada.Numerics.Real_Arrays package implements operations for the **Float** type. Similar packages are available for **Long_Float** and **Long_Long_Float** types. In addition, the Ada.Numerics.**Generic_Real_Arrays** package is a generic version that you can instantiate with custom floating-point types. For example, the **Real_Arrays** package can be defined as follows:

```
package Real_Arrays is new  
  Ada.Numerics.Generic_Real_Arrays (Float);
```

CHAPTER
TWENTYFOUR

APPENDICES

24.1 Appendix A: Generic Formal Types

The following tables contain examples of available formal types for generics:

Formal type	Actual type
Incomplete type Format: <code>type T;</code>	Any type
Discrete type Format: <code>type T is (<>);</code>	Any integer, modular or enumeration type
Range type Format: <code>type T is range <>;</code>	Any signed integer type
Modular type Format: <code>type T is mod <>;</code>	Any modular type
Floating-point type Format: <code>type T is digits <>;</code>	Any floating-point type
Binary fixed-point type Format: <code>type T is delta <>;</code>	Any binary fixed-point type
Decimal fixed-point type Format: <code>type T is delta <> digits <>;</code>	Any decimal fixed-point type
Definite nonlimited private type Format: <code>type T is private;</code>	Any nonlimited, definite type
Nonlimited Private type with discriminant Format: <code>type T (D : DT) is private;</code>	Any nonlimited type with discriminant
Access type Format: <code>type A is access T;</code>	Any access type for type T
Definite derived type Format: <code>type T is new B;</code>	Any concrete type derived from base type B
Limited private type Format: <code>type T is limited private;</code>	Any definite type, limited or not
Incomplete tagged type Format: <code>type T is tagged;</code>	Any concrete, definite, tagged type
Definite tagged private type Format: <code>type T is tagged private;</code>	Any concrete, definite, tagged type
Definite tagged limited private type Format: <code>type T is tagged limited private;</code>	Any concrete definite tagged type, limited or not
Definite abstract tagged private type Format: <code>type T is abstract tagged private;</code>	Any nonlimited, definite tagged type, abstract or concrete

continues on next page

Table 1 – continued from previous page

Formal type	Actual type
Definite abstract tagged limited private type Format: <code>type T is abstract tagged limited private;</code>	Any definite tagged type, limited or not, abstract or concrete
Definite derived tagged type Format: <code>type T is new B with private;</code>	Any concrete tagged type derived from base type B
Definite abstract derived tagged type Format: <code>type T is abstract new B with private;</code>	Any tagged type derived from base type B abstract or concrete
Array type Format: <code>type A is array (R) of T;</code>	Any array type with range R containing elements of type T
Interface type Format: <code>type T is interface;</code>	Any interface type T
Limited interface type Format: <code>type T is limited interface;</code>	Any limited interface type T
Task interface type Format: <code>type T is task interface;</code>	Any task interface type T
Synchronized interface type Format: <code>type T is synchronized interface;</code>	Any synchronized interface type T
Protected interface type Format: <code>type T is protected interface;</code>	Any protected interface type T
Derived interface type Format: <code>type T is new B and I with private;</code>	Any type T derived from base type B and interface I
Derived type with multiple interfaces Format: <code>type T is new B and I1 and I2 with private;</code>	Any type T derived from base type B and interfaces I1 and I2
Abstract derived interface type Format: <code>type T is abstract new B and I with private;</code>	Any type T derived from abstract base type B and interface I
Limited derived interface type Format: <code>type T is limited new B and I with private;</code>	Any type T derived from limited base type B and limited interface I
Abstract limited derived interface type Format: <code>type T is abstract limited new B and I with private;</code>	Any type T derived from abstract limited base type B and limited interface I
Synchronized interface type Format: <code>type T is synchronized new SI with private;</code>	Any type T derived from synchronized interface SI
Abstract synchronized interface type Format: <code>type T is abstract synchronized new SI with private;</code>	Any type T derived from synchronized interface SI

24.1.1 Indefinite version

Many of the examples above can be used for formal indefinite types:

Formal type	Actual type
Indefinite incomplete type Format: <code>type T (<>);</code>	Any type
Indefinite nonlimited private type Format: <code>type T (<>) is private;</code>	Any nonlimited type indefinite or definite
Indefinite limited private type Format: <code>type T (<>) is limited private;</code>	Any type, limited or not, indefinite or definite
Incomplete indefinite tagged private type Format: <code>type T (<>) is tagged;</code>	Any concrete tagged type, indefinite or definite
Indefinite tagged private type Format: <code>type T (<>) is tagged private;</code>	Any concrete, nonlimited tagged type, indefinite or definite
Indefinite tagged limited private type Format: <code>type T (<>) is tagged limited private;</code>	Any concrete tagged type, limited or not, indefinite or definite
Indefinite abstract tagged private type Format: <code>type T (<>) is abstract tagged private;</code>	Any nonlimited tagged type, indefinite or definite, abstract or concrete
Indefinite abstract tagged limited private type Format: <code>type T (<>) is abstract tagged limited private;</code>	Any tagged type, limited or not, indefinite or definite abstract or concrete
Indefinite derived tagged type Format: <code>type T (<>) is new B with private;</code>	Any tagged type derived from base type B, indefinite or definite
Indefinite abstract derived tagged type Format: <code>type T (<>) is abstract new B with private;</code>	Any tagged type derived from base type B, indefinite or definite abstract or concrete

The same examples could also contain discriminants. In this case, `(<>)` is replaced by a list of discriminants, e.g.: `(D: DT)`.

24.2 Appendix B: Containers

The following table shows all containers available in Ada, including their versions (standard, bounded, unbounded, indefinite):

Category	Container	Std	Bounded	Unbounded	Indefinite
Vector	Vectors	Y	Y		Y
List	Doubly Linked Lists	Y	Y		Y
Map	Hashed Maps	Y	Y		Y
Map	Ordered Maps	Y	Y		Y
Set	Hashed Sets	Y	Y		Y
Set	Ordered Sets	Y	Y		Y
Tree	Multiway Trees	Y	Y		Y
Generic	Holders				Y
Queue	Synchronized Queue Interfaces	Y			
Queue	Synchronized Queues		Y	Y	
Queue	Priority Queues		Y	Y	

Note: To get the correct container name, replace the whitespace by `_` in the names above.
(For example, Hashed Maps becomes Hashed_Maps.)

The following table presents the prefixing applied to the container name that depends on its version. As indicated in the table, the standard version does not have a prefix associated with it.

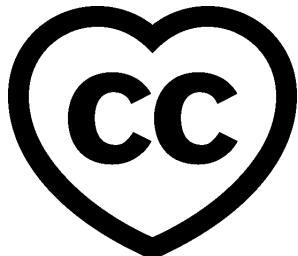
Version	Naming prefix
Std	
Bounded	Bounded_
Unbounded	Unbounded_
Indefinite	Indefinite_

Part II

Introduction To SPARK

Copyright © 2018 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page²⁰](#)



This tutorial is an interactive introduction to the SPARK programming language and its formal verification tools. You will learn the difference between Ada and SPARK and how to use the various analysis tools that come with SPARK.

This document was prepared by Claire Dross and Yannick Moy.

²⁰ <http://creativecommons.org/licenses/by-sa/4.0>

SPARK OVERVIEW

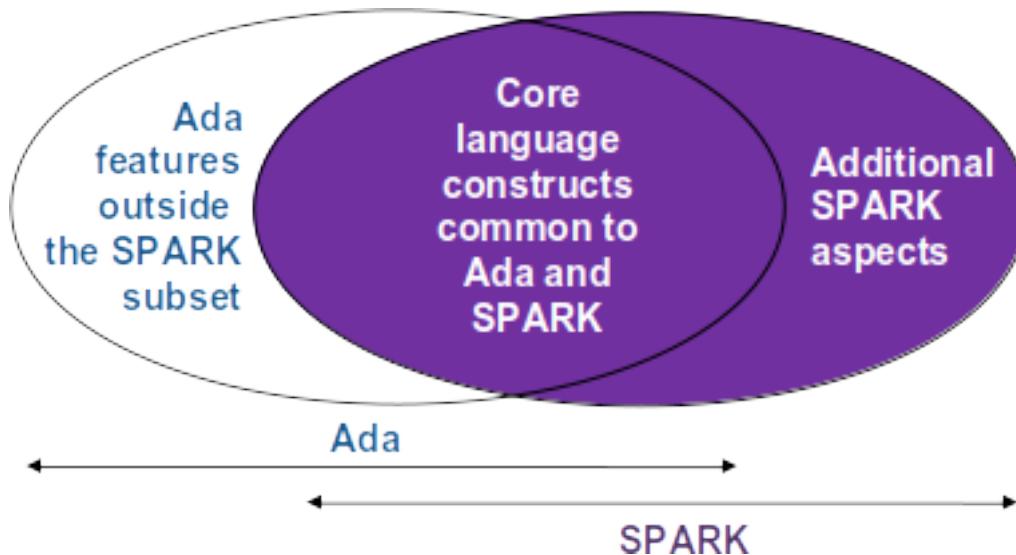
This tutorial is an introduction to the SPARK programming language and its formal verification tools. You need not know any specific programming language (although going over the *Introduction to Ada course* (page 5) first may help) or have experience in formal verification.

25.1 What is it?

SPARK refers to two different things:

- a programming language targeted at functional specification and static verification, and
- a set of development and verification tools for that language.

The SPARK language is based on a subset of the Ada language. Ada is particularly well suited to formal verification since it was designed for critical software development. SPARK builds on that foundation.



Version 2012 of Ada introduced the use of *aspects*, which can be used for subprogram contracts, and version 2014 of SPARK added its own aspects to further aid static analysis.

25.2 What do the tools do?

We start by reviewing static verification of programs, which is verification of the source code performed without compiling or executing it. Verification uses tools that perform static analysis. These can take various forms. They include tools that check types and enforce visibility rules, such as the compiler, in addition to those that perform more complex reasoning, such as abstract interpretation, as done by a tool like [CodePeer²¹](#) from AdaCore. The tools that come with SPARK perform two different forms of static analysis:

- *flow analysis* is the fastest form of analysis. It checks initializations of variables and looks at data dependencies between inputs and outputs of subprograms. It can also find unused assignments and unmodified variables.
- *proof* checks for the absence of runtime errors as well as the conformance of the program with its specifications.

25.3 Key Tools

The tool for formal verification of the SPARK language is called *GNATprove*. It checks for conformance with the SPARK subset and performs flow analysis and proof of the source code. Several other tools support the SPARK language, including both the [GNAT compiler²²](#) and the [GNAT Studio integrated development environment²³](#).

25.4 A trivial example

We start with a simple example of a subprogram in Ada that uses SPARK aspects to specify verifiable subprogram contracts. The subprogram, called Increment, adds 1 to the value of its parameter X:

Listing 1: increment.ads

```
1 procedure Increment
2   (X : in out Integer)
3 with
4   Global  => null,
5   Depends => (X => X),
6   Pre      => X < Integer'Last,
7   Post     => X = X'Old + 1;
```

Listing 2: increment.adb

```
1 procedure Increment
2   (X : in out Integer)
3 is
4 begin
5   X := X + 1;
6 end Increment;
```

Prover output

²¹ <https://www.adacore.com/codepeer>

²² <https://www.adacore.com/gnatpro>

²³ <https://www.adacore.com/gnatpro/toolsuite/gps>

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
increment.adb:5:10: info: overflow check proved
increment.ads:4:03: info: data dependencies proved
increment.ads:5:03: info: flow dependencies proved
increment.ads:7:14: info: postcondition proved
increment.ads:7:24: info: overflow check proved

```

The contracts are written using the Ada *aspect* feature and those shown specify several properties of this subprogram:

- The SPARK Global aspect says that Increment does not read or write any global variables.
- The SPARK Depend aspect is especially interesting for security: it says that the value of the parameter X after the call depends only on the (previous) value of X.
- The Pre and Post aspects of Ada specify functional properties of Increment:
 - Increment is only allowed to be called if the value of X prior to the call is less than `Integer'Last`. This ensures that the addition operation performed in the subprogram body doesn't overflow.
 - Increment does indeed perform an increment of X: the value of X after a call is one greater than its value before the call.

GNATprove can verify all of these contracts. In addition, it verifies that no error can be raised at runtime when executing Increment's body.

25.5 The Programming Language

It's important to understand why there are differences between the SPARK and Ada languages. The aim when designing the SPARK subset of Ada was to create the largest possible subset of Ada that was still amenable to simple specification and sound verification.

The most notable restrictions from Ada are related to exceptions and access types, both of which are known to considerably increase the amount of user-written annotations required for full support. Backwards goto statements and controlled types are also not supported since they introduce non-trivial control flow. The two remaining restrictions relate to side-effects in expressions and aliasing of names, which we now cover in more detail.

25.6 Limitations

25.6.1 No side-effects in expressions

The SPARK language doesn't allow side-effects in expressions. In other words, evaluating a SPARK expression must not update any object. This limitation is necessary to avoid unpredictable behavior that depends on order of evaluation, parameter passing mechanisms, or compiler optimizations. The expression for Dummy below is non-deterministic due to the order in which the two calls to F are evaluated. It's therefore not legal SPARK.

Listing 3: show_illegal_ada_code.adb

```

1  procedure Show_Illegal_Ada_Code is
2
3    function F (X : in out Integer) return Integer is

```

(continues on next page)

(continued from previous page)

```

4   Tmp : constant Integer := X;
5 begin
6   X := X + 1;
7   return Tmp;
8 end F;
9
10 Dummy : Integer := 0;
11
12 begin
13   Dummy := F (Dummy) - F (Dummy); -- ??
14 end Show_Illegal_Ada_Code;

```

Build output

```

show_illegal_ada_code.adb:13:28: error: value may be affected by call to "F"
  ↪because order of evaluation is arbitrary
gprbuild: *** compilation phase failed

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
show_illegal_ada_code.adb:13:28: error: value may be affected by call to "F"
  ↪because order of evaluation is arbitrary
gnatprove: error during generation of Global contracts

```

In fact, the code above is not even legal Ada, so the same error is generated by the GNAT compiler. But SPARK goes further and GNATprove also produces an error for the following equivalent code that is accepted by the Ada compiler:

Listing 4: show_illegal_spark_code.adb

```

1 procedure Show_Illegal_SPARK_Code is
2
3   Dummy : Integer := 0;
4
5   function F return Integer is
6     Tmp : constant Integer := Dummy;
7   begin
8     Dummy := Dummy + 1;
9     return Tmp;
10    end F;
11
12 begin
13   Dummy := F - F; -- ??
14 end Show_Illegal_SPARK_Code;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_illegal_spark_code.adb:5:13: error: function with output global "Dummy" is
  ↪not allowed in SPARK
gnatprove: error during analysis of data and information flow

```

The SPARK languages enforces the lack of side-effects in expressions by forbidding side-effects in functions, which include modifications to either parameters or global variables. As a consequence, SPARK forbids functions with **out** or **in out** parameters in addition to functions modifying a global variable. Function F below is illegal in SPARK, while Function Incr might be legal if it doesn't modify any global variables and function Incr_And_Log might be illegal if it modifies global variables to perform logging.

```

function F (X : in out Integer) return Integer;      -- Illegal
function Incr (X : Integer) return Integer;          -- OK?
function Incr_And_Log (X : Integer) return Integer; -- OK?

```

In most cases, you can easily replace these functions by procedures with an **out** parameter that returns the computed value.

When it has access to function bodies, GNATprove verifies that those functions are indeed free from side-effects. Here for example, the two functions `Incr` and `Incr_And_Log` have the same signature, but only `Incr` is legal in SPARK. `Incr_And_Log` isn't: it attempts to update the global variable `Call_Count`.

Listing 5: `side_effects.ads`

```

1 package Side_Effects is
2
3     function Incr (X : Integer) return Integer;          -- OK?
4
5     function Incr_And_Log (X : Integer) return Integer; -- OK?
6
7 end Side_Effects;

```

Listing 6: `side_effects.adb`

```

1 package body Side_Effects is
2
3     function Incr (X : Integer) return Integer
4     is (X + 1); -- OK
5
6     Call_Count : Natural := 0;
7
8     function Incr_And_Log (X : Integer) return Integer is
9     begin
10        Call_Count := Call_Count + 1; -- Illegal
11        return X + 1;
12    end Incr_And_Log;
13
14 end Side_Effects;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
side_effects.ads:5:13: error: function with output global "Call_Count" is not allowed in SPARK
gnatprove: error during analysis of data and information flow

```

25.6.2 No aliasing of names

Another restriction imposed by the SPARK subset concerns [aliasing](#)²⁴. We say that two names are *aliased* if they refer to the same object. There are two reasons why aliasing is forbidden in SPARK:

- It makes verification more difficult because it requires taking into account the fact that modifications to variables with different names may actually update the same object.

²⁴ [https://en.wikipedia.org/wiki/Aliasing_\(computing\)](https://en.wikipedia.org/wiki/Aliasing_(computing))

- Results may seem unexpected from a user point of view. The results of a subprogram call may depend on compiler-specific attributes, such as parameter passing mechanisms, when its parameters are aliased.

Aliasing can occur as part of the parameter passing that occurs in a subprogram call. Functions have no side-effects in SPARK, so aliasing of parameters in function calls isn't problematic; we need only consider procedure calls. When a procedure is called, SPARK verifies that no **out** or **in out** parameter is aliased with either another parameter of the procedure or a global variable modified in the procedure's body.

Procedure `Move_To_Total` is an example where the possibility of aliasing wasn't taken into account by the programmer:

Listing 7: `no_aliasing.adb`

```

1  procedure No_Aliasing is
2
3      Total : Natural := 0;
4
5      procedure Move_To_Total (Source : in out Natural)
6          with Post => Total = Total'Old + Source'Old and Source = 0
7      is
8          begin
9              Total := Total + Source;
10             Source := 0;
11         end Move_To_Total;
12
13     X : Natural := 3;
14
15    begin
16        Move_To_Total (X);           -- OK
17        pragma Assert (Total = 3); -- OK
18        Move_To_Total (Total);     -- flow analysis error
19        pragma Assert (Total = 6); -- runtime error
20    end No_Aliasing;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
no_aliasing.adb:18:19: high: formal parameter "Source" and global "Total" are
aliased (SPARK RM 6.4.2)
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : no_aliasing.adb:19
```

`Move_To_Total` adds the value of its input parameter `Source` to the global variable `Total` and then resets `Source` to 0. The programmer has clearly not taken into account the possibility of an aliasing between `Total` and `Source`. (This sort of error is quite common.)

This procedure itself is valid SPARK. When doing verification, GNATprove assumes, like the programmer did, that there's no aliasing between `Total` and `Source`. To ensure this assumption is valid, GNATprove checks for possible aliasing on every call to `Move_To_Total`. Its final call in procedure `No_Aliasing` violates this assumption, which produces both a message from GNATprove and a runtime error (an assertion violation corresponding to the expected change in `Total` from calling `Move_To_Total`). Note that the postcondition of `Move_To_Total` is not violated on this second call since integer parameters are passed by copy and the postcondition is checked before the copy-back from the formal parameters to the actual arguments.

Aliasing can also occur as a result of using access types (pointers²⁵ in Ada). These are restricted in SPARK so that only benign aliasing is allowed, when both names are only used to read the data. In particular, assignment between access objects operates a transfer of ownership, where the source object loses its permission to read or write the underlying allocated memory.

Procedure Ownership_Transfer is an example of code that is legal in Ada but rejected in SPARK due to aliasing:

Listing 8: ownership_transfer.adb

```

1  procedure Ownership_Transfer is
2    type Int_Ptr is access Integer;
3    X      : Int_Ptr;
4    Y      : Int_Ptr;
5    Dummy  : Integer;
6  begin
7    X     := new Integer'(1);
8    X.all := X.all + 1;
9    Y     := X;
10   Y.all := Y.all + 1;
11   X.all := X.all + 1;  -- illegal
12   X.all := 1;          -- illegal
13   Dummy := X.all;     -- illegal
14 end Ownership_Transfer;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
ownership_transfer.adb:11:06: error: dereference from "X" is not writable
ownership_transfer.adb:11:06: error: object was moved at line 9
ownership_transfer.adb:11:15: error: dereference from "X" is not readable
ownership_transfer.adb:11:15: error: object was moved at line 9
ownership_transfer.adb:12:06: error: dereference from "X" is not writable
ownership_transfer.adb:12:06: error: object was moved at line 9
ownership_transfer.adb:13:15: error: dereference from "X" is not readable
ownership_transfer.adb:13:15: error: object was moved at line 9
gnatprove: error during analysis of data and information flow

```

After the assignment of X to Y, variable X cannot be used anymore to read or write the underlying allocated memory.

Note: For more details on these limitations, see the [SPARK User's Guide²⁶](#).

25.7 Designating SPARK Code

Since the SPARK language is restricted to only allow easily specifiable and verifiable constructs, there are times when you can't or don't want to abide by these limitations over your entire code base. Therefore, the SPARK tools only check conformance to the SPARK subset on code which you identify as being in SPARK.

You do this by using an aspect named SPARK_Mode. If you don't explicitly specify otherwise, SPARK_Mode is *Off*, meaning you can use the complete set of Ada features in that code and that it should not be analyzed by GNATprove. You can change this default either selectively

²⁵ [https://en.wikipedia.org/wiki/Pointer_\(computer_programming\)](https://en.wikipedia.org/wiki/Pointer_(computer_programming))

²⁶ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#language-restrictions

(on some units or subprograms or packages inside units) or globally (using a configuration pragma, which is what we're doing in this tutorial). To allow simple reuse of existing Ada libraries, entities declared in imported units with no explicit SPARK_Mode can still be used from SPARK code. The tool only checks for SPARK conformance on the declaration of those entities which are actually used within the SPARK code.

Here's a common case of using the SPARK_Mode aspect:

```
package P
  with SPARK_Mode => On
is
  -- package spec is IN SPARK, so can be used by SPARK clients
end P;

package body P
  with SPARK_Mode => Off
is
  -- body is NOT IN SPARK, so is ignored by GNATprove
end P;
```

The package P only defines entities whose specifications are in the SPARK subset. However, it wants to use all Ada features in its body. Therefore the body should not be analyzed and has its SPARK_Mode aspect set to *Off*.

You can specify SPARK_Mode in a fine-grained manner on a per-unit basis. An Ada package has four different components: the visible and private parts of its specification and the declarative and statement parts of its body. You can specify SPARK_Mode as being either *On* or *Off* on any of those parts. Likewise, a subprogram has two parts: its specification and its body.

A general rule in SPARK is that once SPARK_Mode has been set to *Off*, it can never be switched *On* again in the same part of a package or subprogram. This prevents setting SPARK_Mode to *On* for subunits of a unit with SPARK_Mode *Off* and switching back to SPARK_Mode *On* for a part of a given unit where it was set to *Off* in a previous part.

Note: For more details on the use of SPARK_Mode, see the [SPARK User's Guide](#)²⁷.

25.8 Code Examples / Pitfalls

25.8.1 Example #1

Here's a package defining an abstract stack type (defined as a private type in SPARK) of Element objects along with some subprograms providing the usual functionalities of stacks. It's marked as being in the SPARK subset.

Listing 9: stack_package.ads

```
1 package Stack_Package
2   with SPARK_Mode => On
3 is
4   type Element is new Natural;
5   type Stack is private;
6
7   function Empty return Stack;
8   procedure Push (S : in out Stack; E : Element);
```

(continues on next page)

²⁷ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/spark_mode.html

(continued from previous page)

```

9   function Pop (S : in out Stack) return Element;
10
11 private
12   type Stack is record
13     Top : Integer;
14     -- ...
15   end record;
16
17 end Stack_Package;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
stack_package.ads:9:13: error: function with "in out" parameter is not allowed in
  SPARK
stack_package.ads:9:13: error: violation of aspect SPARK_Mode at line 2
gnatprove: error during analysis of data and information flow

```

Side-effects in expressions are not allowed in SPARK. Therefore, Pop is not allowed to modify its parameter S.

25.8.2 Example #2

Let's turn to an abstract state machine version of a stack, where the unit provides a single instance of a stack. The content of the stack (global variables Content and Top) is not directly visible to clients. In this stripped-down version, only the function Pop is available to clients. The package spec and body are marked as being in the SPARK subset.

Listing 10: global_stack.ads

```

1 package Global_Stack
2   with SPARK_Mode => On
3 is
4   type Element is new Integer;
5   function Pop return Element;
6
7 end Global_Stack;

```

Listing 11: global_stack.adb

```

1 package body Global_Stack
2   with SPARK_Mode => On
3 is
4   Max : constant Natural := 100;
5   type Element_Array is array (1 .. Max) of Element;
6
7   Content : Element_Array;
8   Top      : Natural;
9
10  function Pop return Element is
11    E : constant Element := Content (Top);
12  begin
13    Top := Top - 1;
14    return E;
15  end Pop;
16
17 end Global_Stack;

```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
global_stack.adb:7:04: warning: variable "Content" is read but never assigned [-
    ↵gnatwv]
global_stack.ads:6:13: error: function with output global "Top" is not allowed in
    ↵SPARK
gnatprove: error during analysis of data and information flow
```

As above, functions should be free from side-effects. Here, Pop updates the global variable Top, which is not allowed in SPARK.

25.8.3 Example #3

We now consider two procedures: Permute and Swap. Permute applies a circular permutation to the value of its three parameters. Swap then uses Permute to swap the value of X and Y.

Listing 12: p.ads

```
1 package P
2     with SPARK_Mode => On
3 is
4     procedure Permute (X, Y, Z : in out Positive);
5     procedure Swap (X, Y : in out Positive);
6 end P;
```

Listing 13: p.adb

```
1 package body P
2     with SPARK_Mode => On
3 is
4     procedure Permute (X, Y, Z : in out Positive) is
5         Tmp : constant Positive := X;
6     begin
7         X := Y;
8         Y := Z;
9         Z := Tmp;
10    end Permute;
11
12    procedure Swap (X, Y : in out Positive) is
13    begin
14        Permute (X, Y, Y);
15    end Swap;
16 end P;
```

Listing 14: test_swap.adb

```
1 with P; use P;
2
3 procedure Test_Swap
4     with SPARK_Mode => On
5 is
6     A : Integer := 1;
7     B : Integer := 2;
8 begin
9     Swap (A, B);
10 end Test_Swap;
```

Build output

```
p.adb:14:19: error: writable actual for "Y" overlaps with actual for "Z"
gprbuild: *** compilation phase failed
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
p.adb:14:19: error: writable actual for "Y" overlaps with actual for "Z"
gnatprove: error during generation of Global contracts
```

Here, the values for parameters Y and Z are aliased in the call to Permute, which is not allowed in SPARK. In fact, in this particular case, this is even a violation of Ada rules so the same error is issued by the Ada compiler.

In this example, we see the reason why aliasing is not allowed in SPARK: since Y and Z are **Positive**, they are passed by copy and the result of the call to Permute depends on the order in which they're copied back after the call.

25.8.4 Example #4

Here, the Swap procedure is used to swap the value of the two record components of R.

Listing 15: p.ads

```
1 package P
2   with SPARK_Mode => On
3 is
4   type Rec is record
5     F1 : Positive;
6     F2 : Positive;
7   end record;
8
9   procedure Swap_Fields (R : in out Rec);
10  procedure Swap (X, Y : in out Positive);
11 end P;
```

Listing 16: p.adb

```
1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Swap (X, Y : in out Positive) is
5     Tmp : constant Positive := X;
6   begin
7     X := Y;
8     Y := Tmp;
9   end Swap;
10
11  procedure Swap_Fields (R : in out Rec) is
12  begin
13    Swap (R.F1, R.F2);
14  end Swap_Fields;
15
16 end P;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
```

This code is correct. The call to Swap is safe: two different components of the same record can't refer to the same object.

25.8.5 Example #5

Here's a slight modification of the previous example using an array instead of a record: Swap_Indexes calls Swap on values stored in the array A.

Listing 17: p.ads

```

1 package P
2   with SPARK_Mode => On
3 is
4   type P_Array is array (Natural range <>) of Positive;
5
6   procedure Swap_Indexes (A : in out P_Array; I, J : Natural);
7   procedure Swap (X, Y : in out Positive);
8 end P;

```

Listing 18: p.adb

```

1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Swap (X, Y : in out Positive) is
5     Tmp : constant Positive := X;
6   begin
7     X := Y;
8     Y := Tmp;
9   end Swap;
10
11  procedure Swap_Indexes (A : in out P_Array; I, J : Natural) is
12    begin
13      Swap (A (I), A (J));
14  end Swap_Indexes;
15
16 end P;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
p.adb:13:13: medium: formal parameters "X" and "Y" might be aliased (SPARK RM 6.4.
  ↗2)
gnatprove: unproved check messages considered as errors

```

GNATprove detects a possible case of aliasing. Unlike the previous example, it has no way of knowing that the two elements A (I) and A (J) are actually distinct when we call Swap. GNATprove issues a check message here instead of an error, giving you the possibility of justifying the message after review (meaning that you've verified manually that this can't, in fact, occur).

25.8.6 Example #6

We now consider a package declaring a type Dictionary, an array containing a word per letter. The procedure Store allows us to insert a word at the correct index in a dictionary.

Listing 19: p.ads

```

1  with Ada.Finalization;
2
3  package P
4      with SPARK_Mode => On
5  is
6      subtype Letter is Character range 'a' .. 'z';
7      type String_Access is new Ada.Finalization.Controlled with record
8          Ptr : access String;
9      end record;
10     type Dictionary is array (Letter) of String_Access;
11
12     procedure Store (D : in out Dictionary; W : String);
13 end P;

```

Listing 20: p.adb

```

1  package body P
2      with SPARK_Mode => On
3  is
4      procedure Store (D : in out Dictionary; W : String) is
5          First_Letter : constant Letter := W (W'First);
6      begin
7          D (First_Letter).Ptr := new String'(W);
8      end Store;
9  end P;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
p.adb:7:07: error: "String_Access" is not allowed in SPARK (due to controlled
    ↴types)
p.adb:7:07: error: violation of aspect SPARK_Mode at line 2
p.adb:7:31: error: borrow or observe of an expression which is not part of stand-
    ↴alone object or parameter is not allowed in SPARK (SPARK RM 3.10(3)))
p.adb:7:31: error: violation of aspect SPARK_Mode at line 2
p.ads:7:09: error: "Controlled" is not allowed in SPARK (due to controlled types)
p.ads:7:09: error: violation of aspect SPARK_Mode at line 4
p.ads:10:04: error: "String_Access" is not allowed in SPARK (due to controlled
    ↴types)
p.ads:10:04: error: violation of aspect SPARK_Mode at line 4
gnatprove: error during analysis of data and information flow

```

This code is not correct: controlled types are not part of the SPARK subset. The solution here is to use SPARK_Mode to separate the definition of String_Access from the rest of the code in a fine grained manner.

25.8.7 Example #7

Here's a new version of the previous example, which we've modified to hide the controlled type inside the private part of package P, using pragma SPARK_Mode (Off) at the start of the private part.

Listing 21: p.ads

```

1  with Ada.Finalization;
2
3  package P
4      with SPARK_Mode => Off
5  is
6      subtype Letter is Character range 'a' .. 'z';
7      type String_Access is private;
8      type Dictionary is array (Letter) of String_Access;
9
10     function New_String_Access (W : String) return String_Access;
11
12    procedure Store (D : in out Dictionary; W : String);
13
14  private
15      pragma SPARK_Mode (Off);
16
17      type String_Access is new Ada.Finalization.Controlled with record
18          Ptr : access String;
19      end record;
20
21      function New_String_Access (W : String) return String_Access is
22          (Ada.Finalization.Controlled with Ptr => new String'(W));
23  end P;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

```

Since the controlled type is defined and used inside of a part of the code ignored by GNAT-prove, this code is correct.

25.8.8 Example #8

Let's put together the new spec for package P with the body of P seen previously.

Listing 22: p.ads

```

1  with Ada.Finalization;
2
3  package P
4      with SPARK_Mode => On
5  is
6      subtype Letter is Character range 'a' .. 'z';
7      type String_Access is private;
8      type Dictionary is array (Letter) of String_Access;
9
10     function New_String_Access (W : String) return String_Access;
11
12    procedure Store (D : in out Dictionary; W : String);
13
14  private

```

(continues on next page)

(continued from previous page)

```

15 pragma SPARK_Mode (Off);
16
17 type String_Access is new Ada.Finalization.Controlled with record
18   Ptr : access String;
19 end record;
20
21 function New_String_Access (W : String) return String_Access is
22   (Ada.Finalization.Controlled with Ptr => new String'(W));
23 end P;

```

Listing 23: p.adb

```

1 package body P
2   with SPARK_Mode => On
3 is
4   procedure Store (D : in out Dictionary; W : String) is
5     First_Letter : constant Letter := W (W'First);
6   begin
7     D (First_Letter) := New_String_Access (W);
8   end Store;
9 end P;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
p.adb:1:01: error: incorrect application of SPARK_Mode at /vagrant/frontend/dist/
  ↵test_output/projects/Courses/Intro_To_Spark/Overview/Example_08/main.adc:12
p.adb:1:01: error: value Off was set for SPARK_Mode on "P" at p.ads:15
p.adb:2:08: error: incorrect use of SPARK_Mode
p.adb:2:08: error: value Off was set for SPARK_Mode on "P" at p.ads:15
gnatprove: error during generation of Global contracts

```

The body of `Store` doesn't actually use any construct that's not in the SPARK subset, but we nevertheless can't set `SPARK_Mode` to `On` for `P`'s body because it has visibility to `P`'s private part, which is not in SPARK, even if we don't use it.

25.8.9 Example #9

Next, we moved the declaration and the body of the procedure `Store` to another package named `Q`.

Listing 24: p.ads

```

1 with Ada.Finalization;
2
3 package P
4   with SPARK_Mode => On
5 is
6   subtype Letter is Character range 'a' .. 'z';
7   type String_Access is private;
8   type Dictionary is array (Letter) of String_Access;
9
10  function New_String_Access (W : String) return String_Access;
11
12 private
13   pragma SPARK_Mode (Off);
14
15 type String_Access is new Ada.Finalization.Controlled with record
16   Ptr : access String;

```

(continues on next page)

(continued from previous page)

```

17  end record;
18
19  function New_String_Access (W : String) return String_Access is
20      (Ada.Finalization.Controlled with Ptr => new String'(W));
21 end P;

```

Listing 25: q.ads

```

1  with P; use P;
2  package Q
3      with SPARK_Mode => On
4  is
5      procedure Store (D : in out Dictionary; W : String);
6  end Q;

```

Listing 26: q.adb

```

1  package body Q
2      with SPARK_Mode => On
3  is
4      procedure Store (D : in out Dictionary; W : String)  is
5          First_Letter : constant Letter := W (W'First);
6      begin
7          D (First_Letter) := New_String_Access (W);
8      end Store;
9  end Q;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

```

And now everything is fine: we've managed to retain the use of the controlled type while having most of our code in the SPARK subset so GNATprove is able to analyze it.

25.8.10 Example #10

Our final example is a package with two functions to search for the value 0 inside an array A. The first raises an exception if 0 isn't found in A while the other simply returns 0 in that case.

Listing 27: p.ads

```

1  package P
2      with SPARK_Mode => On
3  is
4      type N_Array is array (Positive range <>) of Natural;
5      Not_Found : exception;
6
7      function Search_Zero_P (A : N_Array) return Positive;
8
9      function Search_Zero_N (A : N_Array) return Natural;
10 end P;

```

Listing 28: p.adb

```

1  package body P
2      with SPARK_Mode => On

```

(continues on next page)

(continued from previous page)

```

3  is
4      function Search_Zero_P (A : N_Array) return Positive is
5      begin
6          for I in A'Range loop
7              if A (I) = 0 then
8                  return I;
9              end if;
10             end loop;
11             raise Not_Found;
12         end Search_Zero_P;
13
14     function Search_Zero_N (A : N_Array) return Natural
15     with SPARK_Mode => Off is
16     begin
17         return Search_Zero_P (A);
18     exception
19         when Not_Found => return 0;
20     end Search_Zero_N;
21 end P;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
p.adb:11:07: medium: exception might be raised
gnatprove: unproved check messages considered as errors

```

This code is perfectly correct, despite the use of exception handling, because we've carefully isolated this non-SPARK feature in a function body marked with a SPARK_Mode of Off so it's ignored by GNATprove. However, GNATprove tries to show that Not_Found is never raised in Search_Zero_P, producing a message about a possible exception being raised. Looking at Search_Zero_N, it's indeed likely that an exception is meant to be raised in some cases, which means you need to verify that Not_Found is only raised when appropriate using other methods such as peer review or testing.

FLOW ANALYSIS

In this section we present the flow analysis capability provided by the GNATprove tool, a critical tool for using SPARK.

26.1 What does flow analysis do?

Flow analysis concentrates primarily on variables. It models how information flows through them during a subprogram's execution, connecting the final values of variables to their initial values. It analyzes global variables declared at library level, local variables, and formal parameters of subprograms.

Nesting of subprograms creates what we call *scope variables*: variables declared locally to an enclosing unit. From the perspective of a nested subprogram, scope variables look very much like global variables.

Flow analysis is usually fast, roughly as fast as compilation. It detects various types of errors and finds violations of some SPARK legality rules, such as the absence of aliasing and freedom of expressions from side-effects. We discussed these rules in the *SPARK Overview* (page 263).

Flow analysis is *sound*: if it doesn't detect any errors of a type it's supposed to detect, we know for sure there are no such errors.

26.2 Errors Detected

26.2.1 Uninitialized Variables

We now present each class of errors detected by flow analysis. The first is the reading of an uninitialized variable. This is nearly always an error: it introduces non-determinism and breaks the type system because the value of an uninitialized variable may be outside the range of its subtype. For these reasons, SPARK requires every variable to be initialized before being read.

Flow analysis is responsible for ensuring that SPARK code always fulfills this requirement. For example, in the function `Max_Array` shown below, we've neglected to initialize the value of `Max` prior to entering the loop. As a consequence, the value read by the condition of the `if` statement may be uninitialized. Flow analysis detects and reports this error.

Listing 1: `show_uninitialized.ads`

```
1 package Show_Uninitialized is
2
3     type Array_Of_Naturals is array (Integer range <>) of Natural;
```

(continues on next page)

(continued from previous page)

```
4   function Max_Array (A : Array_Of_Naturals) return Natural;
5
6 end Show_Uninitialized;
```

Listing 2: show_uninitialized.adb

```
1 package body Show_Uninitialized is
2
3   function Max_Array (A : Array_Of_Naturals) return Natural is
4     Max : Natural;
5   begin
6     for I in A'Range loop
7       if A (I) > Max then -- Here Max may not be initialized
8         Max := A (I);
9       end if;
10      end loop;
11      return Max;
12    end Max_Array;
13
14 end Show_Uninitialized;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_uninitialized.adb:7:21: warning: "Max" may be referenced before it has a ↳
value [enabled by default]
show_uninitialized.adb:7:21: medium: "Max" might not be initialized
show_uninitialized.adb:11:14: medium: "Max" might not be initialized
gnatprove: unproved check messages considered as errors
```

Note: For more details on how flow analysis verifies data initialization, see the SPARK User's Guide²⁸.

26.2.2 Ineffective Statements

Ineffective statements are different than dead code: they're executed, and often even modify the value of variables, but have no effect on any of the subprogram's visible outputs: parameters, global variables or the function result. Ineffective statements should be avoided because they make the code less readable and more difficult to maintain.

More importantly, they're often caused by errors in the program: the statement may have been written for some purpose, but isn't accomplishing that purpose. These kinds of errors can be difficult to detect in other ways.

For example, the subprograms Swap1 and Swap2 shown below don't properly swap their two parameters X and Y. This error caused a statement to be ineffective. That ineffective statement is not an error in itself, but flow analysis produces a warning since it can be indicative of an error, as it is here.

²⁸ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/language_restrictions.html#data-initialization-policy

Listing 3: show_ineffective_statements.ads

```

1 package Show_Ineffective_Statements is
2
3   type T is new Integer;
4
5   procedure Swap1 (X, Y : in out T);
6   procedure Swap2 (X, Y : in out T);
7
8 end Show_Ineffective_Statements;

```

Listing 4: show_ineffective_statements.adb

```

1 package body Show_Ineffective_Statements is
2
3   procedure Swap1 (X, Y : in out T) is
4     Tmp : T;
5   begin
6     Tmp := X; -- This statement is ineffective
7     X := Y;
8     Y := X;
9   end Swap1;
10
11  Tmp : T := 0;
12
13  procedure Swap2 (X, Y : in out T) is
14    Temp : T := X; -- This variable is unused
15  begin
16    X := Y;
17    Y := Tmp;
18  end Swap2;
19
20 end Show_Ineffective_Statements;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_ineffective_statements.adb:4:07: warning: variable "Tmp" is assigned but
  ↵never read [-gnatwm]
show_ineffective_statements.adb:6:07: warning: possibly useless assignment to "Tmp"
  ↵", value might not be referenced [-gnatwm]
show_ineffective_statements.adb:6:11: warning: unused assignment
show_ineffective_statements.adb:11:04: warning: "Tmp" is not modified, could be
  ↵declared constant [-gnatwk]
show_ineffective_statements.adb:14:07: warning: variable "Temp" is not referenced
  ↵[-gnatwu]
show_ineffective_statements.ads:5:21: warning: unused initial value of "X"
show_ineffective_statements.ads:6:21: warning: unused initial value of "X"

```

So far, we've seen examples where flow analysis warns about ineffective statements and unused variables.

26.2.3 Incorrect Parameter Mode

Parameter modes are an important part of documenting the usage of a subprogram and affect the code generated for that subprogram. Flow analysis checks that each specified parameter mode corresponds to the usage of that parameter in the subprogram's body. It checks that an **in** parameter is never modified, either directly or through a subprogram call, checks that the initial value of an **out** parameter is never read in the subprogram (since it may not be defined on subprogram entry), and warns when an **in out** parameter isn't modified or when its initial value isn't used. All of these may be signs of an error.

We see an example below. The subprogram Swap is incorrect and GNATprove warns about an input which isn't read:

Listing 5: show_incorrect_param_mode.ads

```

1 package Show_Incorrect_Param_Mode is
2
3     type T is new Integer;
4
5     procedure Swap (X, Y : in out T);
6
7 end Show_Incorrect_Param_Mode;
```

Listing 6: show_incorrect_param_mode.adb

```

1 package body Show_Incorrect_Param_Mode is
2
3     procedure Swap (X, Y : in out T) is
4         Tmp : T := X;
5     begin
6         Y := X;    -- The initial value of Y is not used
7         X := Tmp; -- Y is computed to be an out parameter
8     end Swap;
9
10 end Show_Incorrect_Param_Mode;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_incorrect_param_mode.adb:4:07: warning: "Tmp" is not modified, could be
→ declared constant [-gnatwk]
show_incorrect_param_mode.ads:5:23: warning: unused initial value of "Y"
```

In SPARK, unlike Ada, you should declare an **out** parameter to be **in out** if it's not modified on every path, in which case its value may depend on its initial value. SPARK is stricter than Ada to allow more static detection of errors. This table summarizes SPARK's valid parameter modes as a function of whether reads and writes are done to the parameter.

Initial value read	Written on some path	Written on every path	Parameter mode
X			in
X	X		in out
X		X	in out
	X		in out
		X	out

26.3 Additional Verifications

26.3.1 Global Contracts

So far, none of the verifications we've seen require you to write any additional annotations. However, flow analysis also checks flow annotations that you write. In SPARK, you can specify the set of global and scoped variables accessed or modified by a subprogram. You do this using a contract named `Global`.

When you specify a `Global` contract for a subprogram, flow analysis checks that it's both correct and complete, meaning that no variables other than those stated in the contract are accessed or modified, either directly or through a subprogram call, and that all those listed are accessed or modified. For example, we may want to specify that the function `Get_Value_0f_X` reads the value of the global variable `X` and doesn't access any other global variable. If we do this through a comment, as is usually done in other languages, GNATprove can't verify that the code complies with this specification:

```
package Show_Global_Contracts is
    X : Natural := 0;

    function Get_Value_0f_X return Natural;
    -- Get_Value_0f_X reads the value of the global variable X

end Show_Global_Contracts;
```

You write global contracts as part of the subprogram specification. In addition to their value in flow analysis, they also provide useful information to users of a subprogram. The value you specify for the `Global` aspect is an aggregate-like list of global variable names, grouped together according to their mode.

In the example below, the procedure `Set_X_To_Y_Plus_Z` reads both `Y` and `Z`. We indicate this by specifying them as the value for `Input`. It also writes `X`, which we specify using `Output`. Since `Set_X_To_X_Plus_Y` both writes `X` and reads its initial value, `X`'s mode is `In_Out`. Like parameters, if no mode is specified in a `Global` aspect, the default is `Input`. We see this in the case of the declaration of `Get_Value_0f_X`. Finally, if a subprogram, such as `Incr_Parameter_X`, doesn't reference any global variables, you set the value of the global contract to `null`.

Listing 7: show_global_contracts.ads

```
1 package Show_Global_Contracts is
2
3     X, Y, Z : Natural := 0;
4
5     procedure Set_X_To_Y_Plus_Z with
6         Global => (Input => (Y, Z), -- reads values of Y and Z
7                     Output => X);      -- modifies value of X
8
9     procedure Set_X_To_X_Plus_Y with
10        Global => (Input => Y,   -- reads value of Y
11                     In_Out => X); -- modifies value of X and
12                               -- also reads its initial value
13
14     function Get_Value_0f_X return Natural with
15         Global => X;      -- reads the value of the global variable X
16
17     procedure Incr_Parameter_X (X : in out Natural) with
18         Global => null; -- do not reference any global variable
```

(continues on next page)

(continued from previous page)

```
19
20 end Show_Global_Contracts;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
```

Note: For more details on global contracts, see the SPARK User's Guide²⁹.

26.3.2 Depends Contracts

You may also supply a Depends contract for a subprogram to specify dependencies between its inputs and outputs. These dependencies include not only global variables but also parameters and the function's result. When you supply a Depends contract for a subprogram, flow analysis checks that it's correct and complete, that is, for each dependency you list, the variable depends on those listed and on no others.

For example, you may want to say that the new value of each parameter of Swap, shown below, depends only on the initial value of the other parameter and that the value of X after the return of Set_X_To_Zero doesn't depend on any global variables. If you indicate this through a comment, as you often do in other languages, GNATprove can't verify that this is actually the case.

```
package Show_Depends_Contracts is
    type T is new Integer;
    procedure Swap (X, Y : in out T);
        -- The value of X (resp. Y) after the call depends only
        -- on the value of Y (resp. X) before the call
        X : Natural;
        procedure Set_X_To_Zero;
        -- The value of X after the call depends on no input
end Show_Depends_Contracts;
```

Like Global contracts, you specify a Depends contract in subprogram declarations using an aspect. Its value is a list of one or more dependency relations between the outputs and inputs of the subprogram. Each relation is represented as two lists of variable names separated by an arrow. On the left of each arrow are variables whose final value depends on the initial value of the variables you list on the right.

For example, here we indicate that the final value of each parameter of Swap depends only on the initial value of the other parameter. If the subprogram is a function, we list its result as an output, using the Result attribute, as we do for Get_Value_0f_X below.

Listing 8: show_depends_contracts.ads

```
1 package Show_Depends_Contracts is
2     type T is new Integer;
3
```

(continues on next page)

²⁹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#data-dependencies

(continued from previous page)

```

5   X, Y, Z : T := 0;
6
7   procedure Swap (X, Y : in out T) with
8     Depends => (X => Y,
9                   -- X depends on the initial value of Y
10                  Y => X);
11                   -- Y depends on the initial value of X
12
13  function Get_Value_Of_X return T with
14    Depends => (Get_Value_Of_X'Result => X);
15                   -- result depends on the initial value of X
16
17  procedure Set_X_To_Y_Plus_Z with
18    Depends => (X => (Y, Z));
19                   -- X depends on the initial values of Y and Z
20
21  procedure Set_X_To_X_Plus_Y with
22    Depends => (X =>+ Y);
23                   -- X depends on Y and X's initial value
24
25  procedure Do_Nothing (X : T) with
26    Depends => (null => X);
27                   -- no output is affected by X
28
29  procedure Set_X_To_Zero with
30    Depends => (X => null);
31                   -- X depends on no input
32
33 end Show_Depends_Contracts;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

```

Often, the final value of a variable depends on its own initial value. You can specify this in a concise way using the + character, as we did in the specification of Set_X_To_X_Plus_Y above. If there's more than one variable on the left of the arrow, a + means each variables depends on itself, not that they all depend on each other. You can write the corresponding dependency with ($=>$ +) or without ($=>+$) whitespace.

If you have a program where an input isn't used to compute the final value of any output, you express that by writing `null` on the left of the dependency relation, as we did for the `Do_Nothing` subprogram above. You can only write one such dependency relation, which lists all unused inputs of the subprogram, and it must be written last. Such an annotation also silences flow analysis' warning about unused parameters. You can also write `null` on the right of a dependency relation to indicate that an output doesn't depend on any input. We do that above for the procedure `Set_X_To_Zero`.

Note: For more details on depends contracts, see the [SPARK User's Guide](#)³⁰.

³⁰ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#flow-dependencies

26.4 Shortcomings

26.4.1 Modularity

Flow analysis is sound, meaning that if it doesn't output a message on some analyzed SPARK code, you can be assured that none of the errors it tests for can occur in that code. On the other hand, flow analysis often issues messages when there are, in fact, no errors. The first, and probably most common reason for this relates to modularity.

To scale flow analysis to large projects, verifications are usually done on a per-subprogram basis, including detection of uninitialized variables. To analyze this modularly, flow analysis needs to assume the initialization of inputs on subprogram entry and modification of outputs during subprogram execution. Therefore, each time a subprogram is called, flow analysis checks that global and parameter inputs are initialized and each time a subprogram returns, it checks that global and parameter outputs were modified.

This can produce error messages on perfectly correct subprograms. An example is `Set_X_To_Y_Plus_Z` below, which only sets its `out` parameter `X` when `Overflow` is `False`.

Listing 9: `set_x_to_y_plus_z.adb`

```

1  procedure Set_X_To_Y_Plus_Z
2    (Y, Z      : Natural;
3     X       : out Natural;
4     Overflow : out Boolean)
5  is
6  begin
7    if Natural'Last - Z < Y then
8      Overflow := True; -- X should be initialized on every path
9    else
10      Overflow := False;
11      X := Y + Z;
12    end if;
13  end Set_X_To_Y_Plus_Z;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
set_x_to_y_plus_z.adb:3:04: medium: "X" might not be initialized in "Set_X_To_Y_
↪Plus_Z" [reason for check: OUT parameter should be initialized on return] ↩
↪[possible fix: initialize "X" on all paths or make "X" an IN OUT parameter]
gnatprove: unproved check messages considered as errors
```

The message means that flow analysis wasn't able to verify that the program didn't read an uninitialized variable. To solve this problem, you can either set `X` to a dummy value when there's an overflow or manually verify that `X` is never used after a call to `Set_X_To_Y_Plus_Z` that returned `True` as the value of `Overflow`.

26.4.2 Composite Types

Another common cause of false alarms is caused by the way flow analysis handles composite types. Let's start with arrays.

Flow analysis treats an entire array as single object instead of one object per element, so it considers modifying a single element to be a modification of the array as a whole. Obviously, this makes reasoning about which global variables are accessed less precise and hence the dependencies of those variables are also less precise. This also affects the ability to accurately detect reads of uninitialized data.

It's sometimes impossible for flow analysis to determine if an entire array object has been initialized. For example, after we write code to initialize every element of an unconstrained array A in chunks, we may still receive a message from flow analysis claiming that the array isn't initialized. To resolve this issue, you can either use a simpler loop over the full range of the array, or (even better) an aggregate assignment, or, if that's not possible, verify initialization of the object manually.

Listing 10: show_composite_types_shortcoming.ads

```

1 package Show_Composite_Types_Shortcoming is
2
3     type T is array (Natural range <>) of Integer;
4
5     procedure Init_Chunks (A : out T);
6     procedure Init_Loop (A : out T);
7     procedure Init_Aggregate (A : out T);
8
9 end Show_Composite_Types_Shortcoming;
```

Listing 11: show_composite_types_shortcoming.adb

```

1 package body Show_Composite_Types_Shortcoming is
2
3     procedure Init_Chunks (A : out T) is
4 begin
5         A (A'First) := 0;
6         for I in A'First + 1 .. A'Last loop
7             A (I) := 0;
8         end loop;
9         -- flow analysis doesn't know that A is initialized
10    end Init_Chunks;
11
12    procedure Init_Loop (A : out T) is
13 begin
14        for I in A'Range loop
15            A (I) := 0;
16        end loop;
17        -- flow analysis knows that A is initialized
18    end Init_Loop;
19
20    procedure Init_Aggregate (A : out T) is
21 begin
22        A := (others => 0);
23        -- flow analysis knows that A is initialized
24    end Init_Aggregate;
25
26 end Show_Composite_Types_Shortcoming;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_composite_types_shortcoming.ads:5:27: medium: "A" might not be initialized in
→ "Init_Chunks" [reason for check: OUT parameter should be fully initialized on
→ return] [possible fix: initialize "A" on all paths, make "A" an IN OUT parameter
→ or annotate it with aspect Relaxed_Initialization]
gnatprove: unproved check messages considered as errors
```

Flow analysis is more precise on record objects because it tracks the value of each component of a record separately within a single subprogram. So when a record object is initialized by successive assignments of its components, flow analysis knows that the entire object is initialized. However, record objects are still treated as single objects when analyzed as an input or output of a subprogram.

Listing 12: show_record_flow_analysis.ads

```
1 package Show_Record_Flow_Analysis is
2
3     type Rec is record
4         F1 : Natural;
5         F2 : Natural;
6     end record;
7
8     procedure Init (R : out Rec);
9
10    end Show_Record_Flow_Analysis;
```

Listing 13: show_record_flow_analysis.adb

```
1 package body Show_Record_Flow_Analysis is
2
3     procedure Init (R : out Rec) is
4     begin
5         R.F1 := 0;
6         R.F2 := 0;
7         -- R is initialized
8     end Init;
9
10    end Show_Record_Flow_Analysis;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_record_flow_analysis.adb:8:20: info: initialization of "R" proved
```

Flow analysis complains when a procedure call initializes only some components of a record object. It'll notify you of uninitialized components, as we see in subprogram `Init_F2` below.

Listing 14: show_record_flow_analysis.ads

```
1 package Show_Record_Flow_Analysis is
2
3     type Rec is record
4         F1 : Natural;
5         F2 : Natural;
6     end record;
7
8     procedure Init (R : out Rec);
9     procedure Init_F2 (R : in out Rec);
```

(continues on next page)

(continued from previous page)

```
11 end Show_Record_Flow_Analysis;
```

Listing 15: show_record_flow_analysis.adb

```
1 package body Show_Record_Flow_Analysis is
2
3     procedure Init_F2
4         (R : in out Rec) is
5     begin
6         R.F2 := 0;
7     end Init_F2;
8
9     procedure Init (R : out Rec) is
10    begin
11        R.F1 := 0;
12        Init_F2 (R); -- R should be initialized before this call
13    end Init;
14
15 end Show_Record_Flow_Analysis;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_record_flow_analysis.adb:12:16: high: "R.F2" is not initialized
gnatprove: unproved check messages considered as errors
```

26.4.3 Value Dependency

Flow analysis is not value-dependent: it never reasons about the values of expressions, only whether they have been set to some value or not. As a consequence, if some execution path in a subprogram is impossible, but the impossibility can only be determined by looking at the values of expressions, flow analysis still considers that path feasible and may emit messages based on it believing that execution along such a path is possible.

For example, in the version of `Absolute_Value` below, flow analysis computes that `R` is uninitialized on a path that enters neither of the two conditional statements. Because it doesn't consider values of expressions, it can't know that such a path is impossible.

Listing 16: absolute_value.adb

```
1 procedure Absolute_Value
2     (X : Integer;
3      R : out Natural)
4 is
5 begin
6     if X < 0 then
7         R := -X;
8     end if;
9     if X >= 0 then
10        R := X;
11    end if;
12    -- flow analysis doesn't know that R is initialized
13 end Absolute_Value;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
absolute_value.adb:3:04: medium: "R" might not be initialized in "Absolute_Value"
  ↪[reason for check: OUT parameter should be initialized on return] [possible fix:
    ↪initialize "R" on all paths or make "R" an IN OUT parameter]
gnatprove: unproved check messages considered as errors
```

To avoid this problem, you should make the control flow explicit, as in this second version of `Absolute_Value`:

Listing 17: `absolute_value.adb`

```
1 procedure Absolute_Value
2   (X :      Integer;
3    R : out Natural)
4 is
5 begin
6   if X < 0 then
7     R := -X;
8   else
9     R := X;
10  end if;
11  -- flow analysis knows that R is initialized
12 end Absolute_Value;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
```

26.4.4 Contract Computation

The final cause of unexpected flow messages that we'll discuss also comes from inaccuracy in computations of contracts. As we explained earlier, both Global and Depends contracts are optional, but GNATprove uses their data for some of its analysis.

For example, flow analysis can't detect reads from uninitialized variables without knowing the set of variables accessed. It needs to analyze and check both the Depends contracts you wrote for a subprogram and those you wrote for callers of that subprogram. Since each flow contract on a subprogram depends on the flow contracts of all the subprograms called inside its body, this computation can often be quite time-consuming. Therefore, flow analysis sometimes trades-off the precision of this computation against the time a more precise computation would take.

This is the case for Depends contracts, where flow analysis simply assumes the worst, that each subprogram's output depends on all of that subprogram's inputs. To avoid this assumption, all you have to do is supply contracts when default ones are not precise enough. You may also want to supply Global contracts to further speed up flow analysis on larger programs.

26.5 Code Examples / Pitfalls

26.5.1 Example #1

The procedure `Search_Array` searches for an occurrence of element `E` in an array `A`. If it finds one, it stores the index of the element in `Result`. Otherwise, it sets `Found` to `False`.

Listing 18: `show_search_array.ads`

```

1 package Show_Search_Array is
2
3 type Array_Of_Positives is array (Natural range <>) of Positive;
4
5 procedure Search_Array
6   (A      : Array_Of_Positives;
7    E      : Positive;
8    Result : out Integer;
9    Found  : out Boolean);
10
11 end Show_Search_Array;
```

Listing 19: `show_search_array.adb`

```

1 package body Show_Search_Array is
2
3 procedure Search_Array
4   (A      : Array_Of_Positives;
5    E      : Positive;
6    Result : out Integer;
7    Found  : out Boolean) is
8 begin
9   for I in A'Range loop
10     if A (I) = E then
11       Result := I;
12       Found  := True;
13       return;
14     end if;
15   end loop;
16   Found := False;
17 end Search_Array;
18
19 end Show_Search_Array;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_search_array.ads:8:07: medium: "Result" might not be initialized in "Search_
→Array" [reason for check: OUT parameter should be initialized on return] ↴
→[possible fix: initialize "Result" on all paths or make "Result" an IN OUT ↴
→parameter]
gnatprove: unproved check messages considered as errors
```

GNATprove produces a message saying that `Result` is possibly uninitialized on return. There are perfectly legal uses of the function `Search_Array`, but flow analysis detects that `Result` is not initialized on the path that falls through from the loop. Even though this program is correct, you shouldn't ignore the message: it means flow analysis cannot guarantee that `Result` is always initialized at the call site and so assumes any read of `Result` at the call site will read initialized data. Therefore, you should either initialize `Result` when `Found` is false, which silences flow analysis, or verify this assumption at each call site by

other means.

26.5.2 Example #2

To avoid the message previously issued by GNATprove, we modify `Search_Array` to raise an exception when `E` isn't found in `A`:

Listing 20: `show_search_array.ads`

```

1 package Show_Search_Array is
2
3     type Array_Of_Positives is array (Natural range <>) of Positive;
4
5     Not_Found : exception;
6
7     procedure Search_Array
8         (A      : Array_Of_Positives;
9          E      : Positive;
10         Result : out Integer);
11 end Show_Search_Array;
```

Listing 21: `show_search_array.adb`

```

1 package body Show_Search_Array is
2
3     procedure Search_Array
4         (A      : Array_Of_Positives;
5          E      : Positive;
6          Result : out Integer) is
7 begin
8     for I in A'Range loop
9         if A (I) = E then
10             Result := I;
11             return;
12         end if;
13     end loop;
14     raise Not_Found;
15 end Search_Array;
16
17 end Show_Search_Array;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_search_array.adb:14:07: medium: exception might be raised
gnatprove: unproved check messages considered as errors
```

Flow analysis doesn't emit any messages in this case, meaning it can verify that `Result` can't be read in SPARK code while uninitialized. But why is that, since `Result` is still not initialized when `E` is not in `A`? This is because the exception, `Not_Found`, can never be caught within SPARK code (SPAK doesn't allow exception handlers). However, the GNATprove tool also tries to ensure the absence of runtime errors in SPARK code, so tries to prove that `Not_Found` is never raised. When it can't do that here, it produces a different message.

26.5.3 Example #3

In this example, we're using a discriminated record for the result of Search_Array instead of conditionally raising an exception. By using such a structure, the place to store the index at which E was found exists only when E was indeed found. So if it wasn't found, there's nothing to be initialized.

Listing 22: show_search_array.ads

```

1 package Show_Search_Array is
2
3     type Array_Of_Positives is array (Natural range <>) of Positive;
4
5     type Search_Result (Found : Boolean := False) is record
6         case Found is
7             when True =>
8                 Content : Integer;
9                 when False => null;
10            end case;
11        end record;
12
13    procedure Search_Array
14        (A      : Array_Of_Positives;
15         E      : Positive;
16         Result : out Search_Result)
17     with Pre => not Result'Constrained;
18
19 end Show_Search_Array;

```

Listing 23: show_search_array.adb

```

1 package body Show_Search_Array is
2
3     procedure Search_Array
4         (A      : Array_Of_Positives;
5          E      : Positive;
6          Result : out Search_Result) is
7     begin
8         for I in A'Range loop
9             if A (I) = E then
10                 Result := (Found    => True,
11                           Content => I);
12                 return;
13             end if;
14         end loop;
15         Result := (Found => False);
16     end Search_Array;
17
18 end Show_Search_Array;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_search_array.adb:10:20: info: discriminant check proved
show_search_array.adb:15:14: info: discriminant check proved
show_search_array.ads:16:07: info: initialization of "Result" proved

```

This example is correct and flow analysis doesn't issue any message: it can verify both that no uninitialized variables are read in Search_Array's body, and that all its outputs are set on return. We've used the attribute Constrained in the precondition of Search_Array to indicate that the value of the Result in argument can be set to any variant of the record

type Search_Result, specifically to either the variant where E was found and where it wasn't.

26.5.4 Example #4

The function Size_Of_Biggest_Increasing_Sequence is supposed to find all sequences within its parameter A that contain elements with increasing values and returns the length of the longest one. To do this, it calls a nested procedure Test_Index iteratively on all the elements of A. Test_Index checks if the sequence is still increasing. If so, it updates the largest value seen so far in this sequence. If not, it means it's found the end of a sequence, so it computes the size of that sequence and stores it in Size_Of_Seq.

Listing 24: show_biggest_increasing_sequence.ads

```

1 package Show_Biggest_Increasing_Sequence is
2
3     type Array_Of_Positives is array (Integer range <>) of Positive;
4
5     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
6         return Natural;
7
8 end Show_Biggest_Increasing_Sequence;
```

Listing 25: show_biggest_increasing_sequence.adb

```

1 package body Show_Biggest_Increasing_Sequence is
2
3     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
4         return Natural
5     is
6         Max          : Natural;
7         End_Of_Seq   : Boolean;
8         Size_Of_Seq  : Natural;
9         Beginning    : Integer;
10
11        procedure Test_Index (Current_Index : Integer) is
12    begin
13        if A (Current_Index) >= Max then
14            Max := A (Current_Index);
15            End_Of_Seq := False;
16        else
17            Max      := 0;
18            End_Of_Seq := True;
19            Size_Of_Seq := Current_Index - Beginning;
20            Beginning := Current_Index;
21        end if;
22    end Test_Index;
23
24        Biggest_Seq : Natural := 0;
25
26    begin
27        for I in A'Range loop
28            Test_Index (I);
29            if End_Of_Seq then
30                Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
31            end if;
32        end loop;
33        return Biggest_Seq;
34    end Size_Of_Biggest_Increasing_Sequence;
```

(continues on next page)

(continued from previous page)

```
36 end Show_Biggest_Increasing_Sequence;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_biggest_increasing_sequence.adb:13:34: medium: "Max" might not be initialized,
  ↵ in call inlined at show_biggest_increasing_sequence.adb:28
show_biggest_increasing_sequence.adb:19:44: medium: "Beginning" might not be
  ↵ initialized, in call inlined at show_biggest_increasing_sequence.adb:28
show_biggest_increasing_sequence.adb:30:41: medium: "Size_Of_Seq" might not be
  ↵ initialized
gnatprove: unproved check messages considered as errors
```

However, this example is not correct. Flow analysis emits messages for `Test_Index` stating that `Max`, `Beginning`, and `Size_Of_Seq` should be initialized before being read. Indeed, when you look carefully, you see that both `Max` and `Beginning` are missing initializations because they are read in `Test_Index` before being written. As for `Size_Of_Seq`, we only read its value when `End_Of_Seq` is true, so it actually can't be read before being written, but flow analysis isn't able to verify its initialization by using just flow information.

The call to `Test_Index` is automatically inlined by GNATprove, which leads to another messages above. If GNATprove couldn't inline the call to `Test_Index`, for example if it was defined in another unit, the same messages would be issued on the call to `Test_Index`.

26.5.5 Example #5

In the following example, we model permutations as arrays where the element at index `I` is the position of the `I`'th element in the permutation. The procedure `Init` initializes a permutation to the identity, where the `I`'th elements is at the `I`'th position. `Cyclic_Permutation` calls `Init` and then swaps elements to construct a cyclic permutation.

Listing 26: `show_permutation.ads`

```
1 package Show_Permutation is
2
3   type Permutation is array (Positive range <>) of Positive;
4
5   procedure Swap (A    : in out Permutation;
6                  I, J : Positive);
7
8   procedure Init (A : out Permutation);
9
10  function Cyclic_Permutation (N : Natural) return Permutation;
11
12 end Show_Permutation;
```

Listing 27: `show_permutation.adb`

```
1 package body Show_Permutation is
2
3   procedure Swap (A    : in out Permutation;
4                  I, J : Positive)
5   is
6     Tmp : Positive := A (I);
7   begin
8     A (I) := A (J);
9     A (J) := Tmp;
```

(continues on next page)

(continued from previous page)

```

10  end Swap;

11
12  procedure Init (A : out Permutation) is
13  begin
14      A (A'First) := A'First;
15      for I in A'First + 1 .. A'Last loop
16          A (I) := I;
17      end loop;
18  end Init;

19
20  function Cyclic_Permutation (N : Natural) return Permutation is
21      A : Permutation (1 .. N);
22  begin
23      Init (A);
24      for I in A'First .. A'Last - 1 loop
25          Swap (A, I, I + 1);
26      end loop;
27      return A;
28  end Cyclic_Permutation;

29
30  end Show_Permutation;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_permutation.adb:6:07: warning: "Tmp" is not modified, could be declared
  ↵constant [-gnatwk]
show_permutation.ads:8:20: medium: "A" might not be initialized in "Init" [reason
  ↵for check: OUT parameter should be fully initialized on return] [possible fix:
  ↵initialize "A" on all paths, make "A" an IN OUT parameter or annotate it with
  ↵aspect Relaxed_Initialization]
gnatprove: unproved check messages considered as errors

```

This program is correct. However, flow analysis will nevertheless still emit messages because it can't verify that every element of A is initialized by the loop in Init. This message is a false alarm. You can either ignore it or justify it safely.

26.5.6 Example #6

This program is the same as the previous one except that we've changed the mode of A in the specification of Init to `in out` to avoid the message from flow analysis on array assignment.

Listing 28: show_permutation.ads

```

1 package Show_Permutation is
2
3     type Permutation is array (Positive range <>) of Positive;
4
5     procedure Swap (A      : in out Permutation;
6                     I, J : Positive);
7
8     procedure Init (A : in out Permutation);
9
10    function Cyclic_Permutation (N : Natural) return Permutation;
11
12  end Show_Permutation;

```

Listing 29: show_permutation.adb

```

1 package body Show_Permutation is
2
3   procedure Swap (A      : in out Permutation;
4                  I, J : Positive)
5   is
6     Tmp : Positive := A (I);
7   begin
8     A (I) := A (J);
9     A (J) := Tmp;
10    end Swap;
11
12  procedure Init (A : in out Permutation) is
13  begin
14    A (A'First) := A'First;
15    for I in A'First + 1 .. A'Last loop
16      A (I) := I;
17    end loop;
18  end Init;
19
20  function Cyclic_Permutation (N : Natural) return Permutation is
21    A : Permutation (1 .. N);
22  begin
23    Init (A);
24    for I in A'First .. A'Last - 1 loop
25      Swap (A, I, I + 1);
26    end loop;
27    return A;
28  end Cyclic_Permutation;
29
30 end Show_Permutation;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_permutation.adb:6:07: warning: "Tmp" is not modified, could be declared constant [-gnatwk]
show_permutation.adb:23:13: high: "A" is not initialized
gnatprove: unproved check messages considered as errors

```

This program is not correct. Changing the mode of a parameter that should really be `out` to `in out` to silence a false alarm is not a good idea. Not only does this obfuscate the specification of `Init`, but flow analysis emits a message on the procedure where `A` is not initialized, as shown by the message in `Cyclic_Permutation`.

26.5.7 Example #7

`Incr_Step_Function` takes an array `A` as an argument and iterates through `A` to increment every element by the value of `Increment`, saturating at a specified threshold value. We specified a Global contract for `Incr_Until_Threshold`.

Listing 30: show_increments.ads

```

1 package Show_Increments is
2
3   type Array_Of_Positives is array (Natural range <>) of Positive;
4

```

(continues on next page)

(continued from previous page)

```

5   Increment : constant Natural := 10;
6
7   procedure Incr_Step_Function (A : in out Array_Of_Positives);
8
9 end Show_Increments;
```

Listing 31: show_increments.adb

```

1 package body Show_Increments is
2
3   procedure Incr_Step_Function (A : in out Array_Of_Positives) is
4
5     Threshold : Positive := Positive'Last;
6
7     procedure Incr_Until_Threshold (I : Integer) with
8       Global => (Input => Threshold,
9                  In_Out => A);
10
11    procedure Incr_Until_Threshold (I : Integer) is
12    begin
13      if Threshold - Increment <= A (I) then
14        A (I) := Threshold;
15      else
16        A (I) := A (I) + Increment;
17      end if;
18    end Incr_Until_Threshold;
19
20    begin
21      for I in A'Range loop
22        if I > A'First then
23          Threshold := A (I - 1);
24        end if;
25        Incr_Until_Threshold (I);
26      end loop;
27    end Incr_Step_Function;
28
29 end Show_Increments;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_increments.adb:8:09: info: data dependencies proved
```

Everything is fine here. Specifically, the Global contract is correct. It mentions both Threshold, which is read but not written in the procedure, and A, which is both read and written. The fact that A is a parameter of an enclosing unit doesn't prevent us from using it inside the Global contract; it really is global to Incr_Until_Threshold. We didn't mention Increment since it's a static constant.

26.5.8 Example #8

We now go back to the procedure `Test_Index` from [Example #4](#) (page 296) and correct the missing initializations. We want to know if the Global contract of `Test_Index` is correct.

Listing 32: `show_biggest_increasing_sequence.ads`

```

1 package Show_Biggest_Increasing_Sequence is
2
3     type Array_Of_Positives is array (Integer range <>) of Positive;
4
5     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
6         return Natural;
7
8 end Show_Biggest_Increasing_Sequence;
```

Listing 33: `show_biggest_increasing_sequence.adb`

```

1 package body Show_Biggest_Increasing_Sequence is
2
3     function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
4         return Natural
5     is
6         Max      : Natural := 0;
7         End_Of_Seq : Boolean;
8         Size_Of_Seq : Natural := 0;
9         Beginning : Integer := A'First - 1;
10
11    procedure Test_Index (Current_Index : Integer) with
12        Global => (In_Out => (Beginning, Max, Size_Of_Seq),
13                    Output => End_Of_Seq,
14                    Input  => Current_Index)
15    is
16    begin
17        if A (Current_Index) >= Max then
18            Max := A (Current_Index);
19            End_Of_Seq := False;
20        else
21            Max      := 0;
22            End_Of_Seq := True;
23            Size_Of_Seq := Current_Index - Beginning;
24            Beginning := Current_Index;
25        end if;
26    end Test_Index;
27
28    Biggest_Seq : Natural := 0;
29
30    begin
31        for I in A'Range loop
32            Test_Index (I);
33            if End_Of_Seq then
34                Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
35            end if;
36        end loop;
37        return Biggest_Seq;
38    end Size_Of_Biggest_Increasing_Sequence;
39
40 end Show_Biggest_Increasing_Sequence;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
show_biggest_increasing_sequence.adb:14:30: error: global item cannot reference
  parameter of subprogram "Test_Index"
gnatprove: error during generation of Global contracts
```

The contract in this example is not correct: Current_Index is a parameter of Test_Index, so we shouldn't reference it as a global variable. Also, we should have listed variable A from the outer scope as an Input in the Global contract.

26.5.9 Example #9

Next, we change the Global contract of Test_Index into a Depends contract. In general, we don't need both contracts because the set of global variables accessed can be deduced from the Depends contract.

Listing 34: show_biggest_increasing_sequence.ads

```
1 package Show_Biggest_Increasing_Sequence is
2
3   type Array_Of_Positives is array (Integer range <>) of Positive;
4
5   function Size_of_Biggest_Increasing_Sequence (A : Array_of_Positives)
6     return Natural;
7
8 end Show_Biggest_Increasing_Sequence;
```

Listing 35: show_biggest_increasing_sequence.adb

```
1 package body Show_Biggest_Increasing_Sequence is
2
3   function Size_of_Biggest_Increasing_Sequence (A : Array_of_Positives)
4     return Natural
5   is
6     Max      : Natural := 0;
7     End_of_Seq : Boolean;
8     Size_of_Seq : Natural := 0;
9     Beginning : Integer := A'First - 1;
10
11   procedure Test_Index (Current_Index : Integer) with
12     Depends => ((Max, End_of_Seq)      => (A, Current_Index, Max),
13                  (Size_of_Seq, Beginning) =>
14                    + (A, Current_Index, Max, Beginning))
15   is
16   begin
17     if A (Current_Index) >= Max then
18       Max := A (Current_Index);
19       End_of_Seq := False;
20     else
21       Max      := 0;
22       End_of_Seq := True;
23       Size_of_Seq := Current_Index - Beginning;
24       Beginning := Current_Index;
25     end if;
26   end Test_Index;
27
28   Biggest_Seq : Natural := 0;
29
30   begin
31     for I in A'Range loop
```

(continues on next page)

(continued from previous page)

```

32     Test_Index (I);
33     if End_Of_Seq then
34         Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
35     end if;
36     end loop;
37     return Biggest_Seq;
38 end Size_Of_Biggest_Increasing_Sequence;
39
40 end Show_Biggest_Increasing_Sequence;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
show_biggest_increasing_sequence.adb:7:07: info: initialization of "End_Of_Seq" ↴
    → proved
show_biggest_increasing_sequence.adb:11:17: info: initialization of "End_Of_Seq" ↴
    → proved
show_biggest_increasing_sequence.adb:12:09: info: flow dependencies proved

```

This example is correct. Some of the dependencies, such as `Size_Of_Seq` depending on `Beginning`, come directly from the assignments in the subprogram. Since the control flow influences the final value of all of the outputs, the variables that are being read, `A`, `Current_Index`, and `Max`, are present in every dependency relation. Finally, the dependencies of `Size_Of_Eq` and `Beginning` on themselves are because they may not be modified by the subprogram execution.

26.5.10 Example #10

The subprogram `Identity` swaps the value of its parameter two times. Its `Depends` contract says that the final value of `X` only depends on its initial value and likewise for `Y`.

Listing 36: `show_swap.ads`

```

1 package Show_Swap is
2
3     procedure Swap (X, Y : in out Positive);
4
5     procedure Identity (X, Y : in out Positive) with
6         Depends => (X => X,
7                         Y => Y);
8
9 end Show_Swap;

```

Listing 37: `show_swap.adb`

```

1 package body Show_Swap is
2
3     procedure Swap (X, Y : in out Positive) is
4         Tmp : constant Positive := X;
5     begin
6         X := Y;
7         Y := Tmp;
8     end Swap;
9
10    procedure Identity (X, Y : in out Positive) is
11    begin
12        Swap (X, Y);

```

(continues on next page)

(continued from previous page)

```
13     Swap (Y, X);  
14   end Identity;  
15  
16 end Show_Swap;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...  
Phase 2 of 2: analysis of data and information flow ...  
show_swap.adb:13:07: warning: actuals for this call may be in wrong order [-gnatw.  
→p]  
show_swap.ads:6:18: medium: missing dependency "X => Y"  
show_swap.ads:7:18: medium: missing dependency "Y => X"  
gnatprove: unproved check messages considered as errors
```

This code is correct, but flow analysis can't verify the Depends contract of Identity because we didn't supply a Depends contract for Swap. Therefore, flow analysis assumes that all outputs of Swap, X and Y, depend on all its inputs, both X and Y's initial values. To prevent this, we should manually specify a Depends contract for Swap.

PROOF OF PROGRAM INTEGRITY

This section presents the proof capability of GNATprove, a major tool for the SPARK language. We focus here on the simpler proofs that you'll need to write to verify your program's integrity. The primary objective of performing proof of your program's integrity is to ensure the absence of runtime errors during its execution.

The analysis steps discussed here are only sound if you've previously performed *Flow Analysis* (page 281). You shouldn't proceed further if you still have unjustified flow analysis messages for your program.

27.1 Runtime Errors

There's always the potential for errors that aren't detected during compilation to occur during a program's execution. These errors, called runtime errors, are those targeted by GNATprove.

There are various kinds of runtime errors, the most common being references that are out of the range of an array (`buffer overflow`³¹ in Ada), subtype range violations, overflows in computations, and divisions by zero. The code below illustrates many examples of possible runtime errors, all within a single statement. Look at the assignment statement setting the $I + J$ 'th cell of an array A to the value P / Q .

Listing 1: show_runtime_errors.ads

```
1 package Show_Runtime_Errors is
2
3     type Nat_Array is array (Integer range <>) of Natural;
4
5     procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);
6
7 end Show_Runtime_Errors;
```

Listing 2: show_runtime_errors.adb

```
1 package body Show_Runtime_Errors is
2
3     procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
4     begin
5         A (I + J) := P / Q;
6     end Update;
7
8 end Show_Runtime_Errors;
```

Prover output

³¹ https://en.wikipedia.org/wiki/Buffer_overflow

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_runtime_errors.adb:5:12: medium: overflow check might fail, cannot prove
  ↵lower bound for I + J [reason for check: result of addition must fit in a 32-
  ↵bits machine integer] [possible fix: add precondition (if J >= 0 then I <=
  ↵Integer'Last - J else I >= Integer'First - J) to subprogram at show_runtime_
  ↵errors.ads:5]
show_runtime_errors.adb:5:12: medium: array index check might fail [reason for
  ↵check: result of addition must be a valid index into the array] [possible fix:
  ↵add precondition (if J >= 0 then I <= A'Last - J else I >= A'First - J) to
  ↵subprogram at show_runtime_errors.ads:5]
show_runtime_errors.adb:5:22: medium: divide by zero might fail [possible fix: add
  ↵precondition (Q /= 0) to subprogram at show_runtime_errors.ads:5]
show_runtime_errors.adb:5:22: medium: overflow check might fail, cannot prove
  ↵lower bound for P / Q [reason for check: result of division must fit in a 32-
  ↵bits machine integer] [possible fix: add precondition (P / Q in Integer) to
  ↵subprogram at show_runtime_errors.ads:5]
show_runtime_errors.adb:5:22: medium: range check might fail, cannot prove lower
  ↵bound for P / Q [reason for check: result of division must fit in the target
  ↵type of the assignment] [possible fix: add precondition (P / Q in Natural) to
  ↵subprogram at show_runtime_errors.ads:5]
gnatprove: unproved check messages considered as errors

```

There are quite a number of errors that may occur when executing this code. If we don't know anything about the values of I, J, P, and Q, we can't rule out any of those errors.

First, the computation of I + J can overflow, for example if I is `Integer'Last` and J is positive.

```
A (Integer'Last + 1) := P / Q;
```

Next, the sum, which is used as an array index, may not be in the range of the index of the array.

```
A (A'Last + 1) := P / Q;
```

On the other side of the assignment, the division may also overflow, though only in the very special case where P is `Integer'First` and Q is -1 because of the asymmetric range of signed integer types.

```
A (I + J) := Integer'First / -1;
```

The division is also not allowed if Q is 0.

```
A (I + J) := P / 0;
```

Finally, since the array contains natural numbers, it's also an error to store a negative value in it.

```
A (I + J) := 1 / -1;
```

The compiler generates checks in the executable code corresponding to each of those runtime errors. Each check raises an exception if it fails. For the above assignment statement, we can see examples of exceptions raised due to failed checks for each of the different cases above.

```

A (Integer'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : index check failed

```

(continues on next page)

(continued from previous page)

```

A (I + J) := Integer'First / (-1);
-- raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
-- raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
-- raised CONSTRAINT_ERROR : divide by zero

```

These runtime checks are costly, both in terms of program size and execution time. It may be appropriate to remove them if we can statically ensure they aren't needed at runtime, in other words if we can prove that the condition tested for can never occur.

This is where the analysis done by GNATprove comes in. It can be used to demonstrate statically that none of these errors can ever occur at runtime. Specifically, GNATprove logically interprets the meaning of every instruction in the program. Using this interpretation, GNATprove generates a logical formula called a *verification condition* for each check that would otherwise be required by the Ada (and hence SPARK) language.

```

A (Integer'Last + 1) := P / Q;
-- medium: overflow check might fail

A (A'Last + 1) := P / Q;
-- medium: array index check might fail

A (I + J) := Integer'First / (-1);
-- medium: overflow check might fail

A (I + J) := 1 / (-1);
-- medium: range check might fail

A (I + J) := P / 0;
-- medium: divide by zero might fail

```

GNATprove then passes these verification conditions to an automatic prover, stated as conditions that must be true to avoid the error. If every such condition can be validated by a prover (meaning that it can be mathematically shown to always be true), we've been able to prove that no error can ever be raised at runtime when executing that program.

27.2 Modularity

To scale to large programs, GNATprove performs proofs on a per-subprogram basis by relying on preconditions and postconditions to properly summarize the input and output state of each subprogram. More precisely, when verifying the body of a subprogram, GNATprove assumes it knows nothing about the possible initial values of its parameters and of the global variables it accesses except what you state in the subprogram's precondition. If you don't specify a precondition, it can't make any assumptions.

For example, the following code shows that the body of Increment can be successfully verified: its precondition constrains the value of its parameter X to be less than `Integer'Last` so we know the overflow check is always false.

In the same way, when a subprogram is called, GNATprove assumes its `out` and `in out` parameters and the global variables it writes can be modified in any way compatible with their postconditions. For example, since Increment has no postcondition, GNATprove doesn't know that the value of X after the call is always less than `Integer'Last`. Therefore, it can't prove that the addition following the call to Increment can't overflow.

Listing 3: show_modularity.adb

```

1  procedure Show_Modularity is
2
3      procedure Increment (X : in out Integer) with
4          Pre => X < Integer'Last is
5          begin
6              X := X + 1;
7              -- info: overflow check proved
8          end Increment;
9
10     X : Integer;
11 begin
12     X := Integer'Last - 2;
13     Increment (X);
14     -- After the call, GNATprove no longer knows the value of X
15
16     X := X + 1;
17     -- medium: overflow check might fail
18 end Show_Modularity;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_modularity.adb:6:14: info: overflow check proved
show_modularity.adb:10:04: info: initialization of "X" proved
show_modularity.adb:13:04: info: precondition proved
show_modularity.adb:16:04: warning: possibly useless assignment to "X", value ↴
    ↴ might not be referenced [-gnatwm]
show_modularity.adb:16:11: medium: overflow check might fail, cannot prove upper ↴
    ↴ bound for X + 1 [reason for check: result of addition must fit in a 32-bits ↴
    ↴ machine integer] [possible fix: call at line 13 should mention X (for argument ↴
    ↴ X) in a postcondition]
gnatprove: unproved check messages considered as errors

```

27.2.1 Exceptions

There are two cases where GNATprove doesn't require modularity and hence doesn't make the above assumptions. First, local subprograms without contracts can be inlined if they're simple enough and are neither recursive nor have multiple return points. If we remove the contract from Increment, it fits the criteria for inlining.

Listing 4: show_modularity.adb

```

1  procedure Show_Modularity is
2
3      procedure Increment (X : in out Integer) is
4          begin
5              X := X + 1;
6              -- info: overflow check proved, in call inlined at...
7          end Increment;
8
9      X : Integer;
10 begin
11     X := Integer'Last - 2;
12     Increment (X);
13     X := X + 1;
14     -- info: overflow check proved
15 end Show_Modularity;

```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_modularity.adb:5:14: info: overflow check proved, in call inlined at show_
→modularity.adb:12
show_modularity.adb:9:04: info: initialization of "X" proved
show_modularity.adb:13:04: warning: possibly useless assignment to "X", value_
→might not be referenced [-gnatwm]
show_modularity.adb:13:11: info: overflow check proved
```

GNATprove now sees the call to Increment exactly as if the increment on X was done outside that call, so it can successfully verify that neither addition can overflow.

Note: For more details on contextual analysis of subprograms, see the [SPARK User's Guide](#)³².

The other case involves functions. If we define a function as an expression function, with or without contracts, GNATprove uses the expression itself as the postcondition on the result of the function.

In our example, replacing Increment with an expression function allows GNATprove to successfully verify the overflow check in the addition.

Listing 5: show_modularity.adb

```
1  procedure Show_Modularity is
2
3      function Increment (X : Integer) return Integer is
4          (X + 1)
5          -- info: overflow check proved
6          with Pre => X < Integer'Last;
7
8          X : Integer;
9      begin
10         X := Integer'Last - 2;
11         X := Increment (X);
12         X := X + 1;
13         -- info: overflow check proved
14     end Show_Modularity;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_modularity.adb:4:09: info: overflow check proved
show_modularity.adb:8:04: info: initialization of "X" proved
show_modularity.adb:11:09: info: precondition proved
show_modularity.adb:12:04: warning: possibly useless assignment to "X", value_
→might not be referenced [-gnatwm]
show_modularity.adb:12:11: info: overflow check proved
```

Note: For more details on expression functions, see the [SPARK User's Guide](#)³³.

³² https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_write_subprogram_contracts.html#contextual-analysis-of-subprograms-without-contracts

³³ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/specification_features.html#expression-functions

27.3 Contracts

Ada contracts are perfectly suited for formal verification, but are primarily designed to be checked at runtime. When you specify the `-gnata` switch, the compiler generates code that verifies the contracts at runtime. If an Ada contract isn't satisfied for a given subprogram call, the program raises the `Assert_Failure` exception. This switch is particularly useful during development and testing, but you may also retain run-time execution of assertions, and specifically preconditions, during the program's deployment to avoid an inconsistent state.

Consider the incorrect call to `Increment` below, which violates its precondition. One way to detect this error is by compiling the function with assertions enabled and testing it with inputs that trigger the violation. Another way, one that doesn't require guessing the needed inputs, is to run GNATprove.

Listing 6: show_preconditionViolation.adb

```

1  procedure Show_Precondition_Violation is
2
3      procedure Increment (X : in out Integer) with
4          Pre => X < Integer'Last  is
5          begin
6              X := X + 1;
7          end Increment;
8
9      X : Integer;
10
11     begin
12         X := Integer'Last;
13         Increment (X);
14     end Show_Precondition_Violation;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_preconditionViolation.adb:13:04: medium: precondition might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```

raised ADA ASSERTIONS ASSERTION_ERROR : failed precondition from show_precondition_
↳ violation.adb:4
```

Similarly, consider the incorrect implementation of function `Absolute` below, which violates its postcondition. Likewise, one way to detect this error is by compiling the function with assertions enabled and testing with inputs that trigger the violation. Another way, one which again doesn't require finding the inputs needed to demonstrate the error, is to run GNATprove.

Listing 7: show_postconditionViolation.adb

```

1  procedure Show_Postcondition_Violation is
2
3      procedure Absolute (X : in out Integer) with
4          Post => X >= 0 is
5          begin
6              if X > 0 then
7                  X := -X;
```

(continues on next page)

(continued from previous page)

```

8   end if;
9 end Absolute;
10
11 X : Integer;
12
13 begin
14   X := 1;
15   Absolute (X);
16 end Show_Postcondition_Violation;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_postconditionViolation.adb:4:14: medium: postcondition might fail
gnatprove: unproved check messages considered as errors

```

Runtime output

```

raised ADA ASSERTIONS ASSERTION_ERROR : failed postcondition from show_
→postconditionViolation.adb:4

```

The benefits of dynamically checking contracts extends beyond making testing easier. Early failure detection also allows an easier recovery and facilitates debugging, so you may want to enable these checks at runtime to terminate execution before some damaging or hard-to-debug action occurs.

GNATprove statically analyses preconditions and postconditions. It verifies preconditions every time a subprogram is called, which is the runtime semantics of contracts. Postconditions, on the other hand, are verified once as part of the verification of the subprogram's body. For example, GNATprove must wait until Increment is improperly called to detect the precondition violation, since a precondition is really a contract for the caller. On the other hand, it doesn't need Absolute to be called to detect that its postcondition doesn't hold for all its possible inputs.

Note: For more details on pre and postconditions, see the [SPARK User's Guide³⁴](#).

27.3.1 Executable Semantics

Expressions in Ada contracts have the same semantics as Boolean expressions elsewhere, so runtime errors can occur during their computation. To simplify both debugging of assertions and combining testing and static verification, the same semantics are used by GNATprove.

While proving programs, GNATprove verifies that no error can ever be raised during the execution of the contracts. However, you may sometimes find those semantics too heavy, in particular with respect to overflow checks, because they can make it harder to specify an appropriate precondition. We see this in the function Add below.

Listing 8: show_executable_semantics.adb

```

1 procedure Show_Executable_Semantics
2   with SPARK_Mode => On

```

(continues on next page)

³⁴ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#preconditions

(continued from previous page)

```

3   is
4     function Add (X, Y : Integer) return Integer is (X + Y)
5       with Pre => X + Y in Integer;
6
7     X : Integer;
8   begin
9     X := Add (Integer'Last, 1);
10  end Show_Executeable_Semantics;

```

Build output

```

show_executable_semantics.adb:5:24: warning: explicit membership test may be
  ↪optimized away [enabled by default]
show_executable_semantics.adb:5:24: warning: use 'Valid attribute instead [enabled
  ↪by default]
show_executable_semantics.adb:7:04: warning: variable "X" is assigned but never
  ↪read [-gnatwm]
show_executable_semantics.adb:9:04: warning: possibly useless assignment to "X",
  ↪value might not be referenced [-gnatwm]

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_executable_semantics.adb:5:20: medium: overflow check might fail, cannot
  ↪prove lower bound for X + Y [reason for check: result of addition must fit in a
  ↪32-bits machine integer] [possible fix: use pragma Overflow_Mode or switch -
  ↪gnato13 or unit Ada.Numerics.Big_Numerics.Big_Integers]
show_executable_semantics.adb:7:04: warning: variable "X" is assigned but never
  ↪read [-gnatwm]
show_executable_semantics.adb:9:04: warning: possibly useless assignment to "X",
  ↪value might not be referenced [-gnatwm]
show_executable_semantics.adb:9:09: medium: precondition might fail, cannot prove
  ↪upper bound for Add (Integer'Last, 1)
gnatprove: unproved check messages considered as errors

```

Runtime output

```
raised CONSTRAINT_ERROR : show_executable_semantics.adb:5 overflow check failed
```

GNATprove issues a message on this code warning about a possible overflow when computing the sum of X and Y in the precondition. Indeed, since expressions in assertions have normal Ada semantics, this addition can overflow, as you can easily see by compiling and running the code that calls Add with arguments `Integer'Last` and 1.

On the other hand, you sometimes may prefer GNATprove to use the mathematical semantics of addition in contracts while the generated code still properly verifies that no error is ever raised at runtime in the body of the program. You can get this behavior by using the compiler switch `-gnato??` (for example `-gnato13`), which allows you to independently set the overflow mode in code (the first digit) and assertions (the second digit). For both, you can either reduce the number of overflow checks (the value 2), completely eliminate them (the value 3), or preserve the default Ada semantics (the value 1).

Note: For more details on overflow modes, see the SPARK User's Guide³⁵.

³⁵ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/overflow_modes.html

27.3.2 Additional Assertions and Contracts

As we've seen, a key feature of SPARK is that it allows us to state properties to check using assertions and contracts. SPARK supports preconditions and postconditions as well as assertions introduced by the `Assert` pragma.

The SPARK language also includes new contract types used to assist formal verification. The new pragma `Assume` is treated as an assertion during execution but introduces an assumption when proving programs. Its value is a Boolean expression which GNATprove assumes to be true without any attempt to verify that it's true. You'll find this feature useful, but you must use it with great care. Here's an example of using it.

Listing 9: incr.adb

```

1 procedure Incr (X : in out Integer) is
2 begin
3     pragma Assume (X < Integer'Last);
4     X := X + 1;
5 end Incr;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
incr.adb:4:11: info: overflow check proved
```

Note: For more details on pragma `Assume`, see the [SPARK User's Guide](#)³⁶.

The `Contract_Cases` aspect is another construct introduced for GNATprove, but which also acts as an assertion during execution. It allows you to specify the behavior of a subprogram using a disjunction of cases. Each element of a `Contract_Cases` aspect is a *guard*, which is evaluated before the call and may only reference the subprogram's inputs, and a *consequence*. At each call of the subprogram, one and only one guard is permitted to evaluate to `True`. The consequence of that case is a contract that's required to be satisfied when the subprogram returns.

Listing 10: absolute.adb

```

1 procedure Absolute (X : in out Integer) with
2     Pre          => X > Integer'First,
3     Contract_Cases => (X < 0 => X = -X'Old,
4                           X >= 0 => X = X'Old)
5 is
6 begin
7     if X < 0 then
8         X := -X;
9     end if;
10    end Absolute;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
absolute.adb:3:03: info: disjoint contract cases proved
absolute.adb:3:03: info: complete contract cases proved
absolute.adb:3:29: info: contract case proved
absolute.adb:3:36: info: overflow check proved
```

(continues on next page)

³⁶ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/assertion_pragmas.html# pragma-assume

(continued from previous page)

```
absolute.adb:4:29: info: contract case proved
absolute.adb:8:12: info: overflow check proved
```

Similarly to how it analyzes a subprogram's precondition, GNATprove verifies the Contract_Cases only once. It verifies the validity of each consequence (given the truth of its guard) and the disjointness and completeness of the guard conditions (meaning that exactly one guard must be true for each possible set of input values).

Note: For more details on Contract_Cases, see the SPARK User's Guide³⁷.

27.4 Debugging Failed Proof Attempts

GNATprove may report an error while verifying a program for any of the following reasons:

- there might be an error in the program; or
- the property may not be provable as written because more information is required; or
- the prover used by GNATprove may be unable to prove a perfectly valid property.

We spend the remainder of this section discussing the sometimes tricky task of debugging failed proof attempts.

27.4.1 Debugging Errors in Code or Specification

First, let's discuss the case where there's indeed an error in the program. There are two possibilities: the code may be incorrect or, equally likely, the specification may be incorrect. As an example, there's an error in our procedure Incr_Until below which makes its Contract_Cases unprovable.

Listing 11: show_failed_proof_attempt.ads

```
1 package Show_Failed_Proof_Attempt is
2
3     Incremented : Boolean := False;
4
5     procedure Incr_Until (X : in out Natural) with
6         Contract_Cases =>
7             (Incremented => X > X'Old,
8              others        => X = X'Old);
9
10    end Show_Failed_Proof_Attempt;
```

Listing 12: show_failed_proof_attempt.adb

```
1 package body Show_Failed_Proof_Attempt is
2
3     procedure Incr_Until (X : in out Natural) is
4     begin
5         if X < 1000 then
6             X := X + 1;
7             Incremented := True;
```

(continues on next page)

³⁷ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#contract-cases

(continued from previous page)

```

8     else
9         Incremented := False;
10    end if;
11   end Incr_Until;
12
13 end Show_Failed_Proof_Attempt;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_failed_proof_attempt.ads:7:21: medium: contract case might fail
show_failed_proof_attempt.ads:8:21: medium: contract case might fail
gnatprove: unproved check messages considered as errors

```

Since this is an assertion that can be executed, it may help you find the problem if you run the program with assertions enabled on representative sets of inputs. This allows you to find bugs in both the code and its contracts. In this case, testing Incr_Until with an input greater than 1000 raises an exception at runtime.

Listing 13: show_failed_proof_attempt.ads

```

1 package Show_Failed_Proof_Attempt is
2
3     Incremented : Boolean := False;
4
5     procedure Incr_Until (X : in out Natural) with
6         Contract_Cases =>
7             (Incremented => X > X'Old,
8              others        => X = X'Old);
9
10 end Show_Failed_Proof_Attempt;

```

Listing 14: show_failed_proof_attempt.adb

```

1 package body Show_Failed_Proof_Attempt is
2
3     procedure Incr_Until (X : in out Natural) is
4         begin
5             if X < 1000 then
6                 X := X + 1;
7                 Incremented := True;
8             else
9                 Incremented := False;
10            end if;
11        end Incr_Until;
12
13 end Show_Failed_Proof_Attempt;

```

Listing 15: main.adb

```

1 with Show_Failed_Proof_Attempt; use Show_Failed_Proof_Attempt;
2
3 procedure Main is
4     Dummy : Integer;
5 begin
6     Dummy := 0;
7     Incr_Until (Dummy);
8
9     Dummy := 1000;

```

(continues on next page)

(continued from previous page)

```
10   Incr_Until (Dummy);
11 end Main;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_failed_proof_attempt.ads:7:21: medium: contract case might fail
show_failed_proof_attempt.ads:8:21: medium: contract case might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : failed contract case at show_failed_proof_
↪attempt.ads:8
```

The error message shows that the first contract case is failing, which means that Incremented is **True**. However, if we print the value of Incremented before returning, we see that it's **False**, as expected for the input we provided. The error here is that guards of contract cases are evaluated before the call, so our specification is wrong! To correct this, we should either write `X < 1000` as the guard of the first case or use a standard postcondition with an if-expression.

27.4.2 Debugging Cases where more Information is Required

Even if both the code and the assertions are correct, GNATprove may still report that it can't prove a verification condition for a property. This can happen for two reasons:

- The property may be unprovable because the code is missing some assertion. One category of these cases is due to the modularity of the analysis which, as we discussed above, means that GNATprove only knows about the properties of your subprograms that you have explicitly written.
- There may be some information missing in the logical model of the program used by GNATprove.

Let's look at the case where the code and the specification are correct but there's some information missing. As an example, GNATprove finds the postcondition of Increase to be unprovable.

Listing 16: `show_failed_proof_attempt.ads`

```
1 package Show_Failed_Proof_Attempt is
2
3   C : Natural := 100;
4
5   procedure Increase (X : in out Natural) with
6     Post => (if X'Old < C then X > X'Old else X = C);
7
8 end Show_Failed_Proof_Attempt;
```

Listing 17: `show_failed_proof_attempt.adb`

```
1 package body Show_Failed_Proof_Attempt is
2
3   procedure Increase (X : in out Natural) is
4   begin
5     if X < 90 then
```

(continues on next page)

(continued from previous page)

```

6      X := X + 10;
7  elsif X >= C then
8      X := C;
9  else
10     X := X + 1;
11  end if;
12 end Increase;
13
14 end Show_Failed_Proof_Attempt;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_failed_proof_attempt.ads:6:49: medium: postcondition might fail, cannot prove_X = C
gnatprove: unproved check messages considered as errors

```

This postcondition is a conditional. It says that if the parameter (X) is less than a certain value (C), its value will be increased by the procedure while if it's greater, its value will be set to C (saturated). When C has the value 100, the code of Increases adds 10 to the value of X if it was initially less than 90, increments X by 1 if it was between 90 and 99, and sets X to 100 if it was greater or equal to 100. This behavior does satisfy the postcondition, so why is the postcondition not provable?

The values in the counterexample returned by GNATprove in its message gives us a clue: `C = 0 and X = 10 and X'Old = 0`. Indeed, if C is not equal to 100, our reasoning above is incorrect: the values of 0 for C and X on entry indeed result in X being 10 on exit, which violates the postcondition!

We probably didn't expect the value of C to change, or at least not to go below 90. But, in that case, we should have stated so by either declaring C to be constant or by adding a precondition to the Increase subprogram. If we do either of those, GNATprove is able to prove the postcondition.

27.4.3 Debugging Prover Limitations

Finally, there are cases where GNATprove provides a perfectly valid verification condition for a property, but it's nevertheless not proved by the automatic prover that runs in the later stages of the tool's execution. This is quite common. Indeed, GNATprove produces its verification conditions in first-order logic, which is not decidable, especially in combination with the rules of arithmetic. Sometimes, the automatic prover just needs more time. Other times, the prover will abandon the search almost immediately or loop forever without reaching a conclusive answer (either a proof or a counterexample).

For example, the postcondition of our GCD function below — which calculates the value of the GCD of two positive numbers using Euclid's algorithm — can't be verified with GNATprove's default settings.

Listing 18: show_failed_proof_attempt.ads

```

1 package Show_Failed_Proof_Attempt is
2
3   function GCD (A, B : Positive) return Positive with
4     Post =>
5       A mod GCD'Result = 0
6       and B mod GCD'Result = 0;
7
8 end Show_Failed_Proof_Attempt;

```

Listing 19: show_failed_proof_attempt.adb

```

1 package body Show_Failed_Proof_Attempt is
2
3     function GCD (A, B : Positive) return Positive is
4 begin
5     if A > B then
6         return GCD (A - B, B);
7     elsif B > A then
8         return GCD (A, B - A);
9     else
10        return A;
11    end if;
12 end GCD;
13
14 end Show_Failed_Proof_Attempt;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_failed_proof_attempt.ads:5:08: medium: postcondition might fail, cannot prove
  ↵A mod GCD'Result = 0
gnatprove: unproved check messages considered as errors

```

The first thing we try is increasing the amount of time the prover is allowed to spend on each verification condition using the `--timeout` option of GNATprove (e.g., by using the dialog box in GNAT Studio). In this example, increasing it to one minute, which is relatively high, doesn't help. We can also specify an alternative automatic prover — if we have one — using the option `--prover` of GNATprove (or the dialog box). For our postcondition, we tried Alt-Ergo, cvc5, and Z3 without any luck.

Listing 20: show_failed_proof_attempt.ads

```

1 package Show_Failed_Proof_Attempt is
2
3     function GCD (A, B : Positive) return Positive with
4 Post =>
5     A mod GCD'Result = 0
6     and B mod GCD'Result = 0;
7
8 end Show_Failed_Proof_Attempt;

```

Listing 21: show_failed_proof_attempt.adb

```

1 package body Show_Failed_Proof_Attempt is
2
3     function GCD (A, B : Positive) return Positive
4 is
5     Result : Positive;
6 begin
7     if A > B then
8         Result := GCD (A - B, B);
9         pragma Assert ((A - B) mod Result = 0);
10        -- info: assertion proved
11        pragma Assert (B mod Result = 0);
12        -- info: assertion proved
13        pragma Assert (A mod Result = 0);
14        -- medium: assertion might fail
15     elsif B > A then
16         Result := GCD (A, B - A);

```

(continues on next page)

(continued from previous page)

```

17  pragma Assert ((B - A) mod Result = 0);
18  -- info: assertion proved
19  else
20      Result := A;
21  end if;
22  return Result;
23 end GCD;
24
25 end Show_Failed_Proof_Attempt;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_failed_proof_attempt.adb:5:07: info: initialization of "Result" proved
show_failed_proof_attempt.adb:8:27: info: range check proved
show_failed_proof_attempt.adb:9:25: info: assertion proved
show_failed_proof_attempt.adb:9:33: info: division check proved
show_failed_proof_attempt.adb:11:25: info: assertion proved
show_failed_proof_attempt.adb:11:27: info: division check proved
show_failed_proof_attempt.adb:13:25: medium: assertion might fail [possible fix:↳ subprogram at show_failed_proof_attempt.ads:3 should mention A in a precondition]
show_failed_proof_attempt.adb:13:27: info: division check proved
show_failed_proof_attempt.adb:16:30: info: range check proved
show_failed_proof_attempt.adb:17:25: info: assertion proved
show_failed_proof_attempt.adb:17:33: info: division check proved
show_failed_proof_attempt.ads:5:10: info: division check proved
show_failed_proof_attempt.ads:6:12: medium: postcondition might fail, cannot prove↳ B mod GCD'Result = 0
show_failed_proof_attempt.ads:6:14: info: division check proved
gnatprove: unproved check messages considered as errors

```

To better understand the reason for the failure, we added intermediate assertions to simplify the proof and pin down the part that's causing the problem. Adding such assertions is often a good idea when trying to understand why a property is not proved. Here, provers can't verify that if both $A - B$ and B can be divided by $Result$ so can A . This may seem surprising, but non-linear arithmetic, involving, for example, multiplication, modulo, or exponentiation, is a difficult topic for provers and is not handled very well in practice by any of the general-purpose ones like Alt-Ergo, cvc5, or Z3.

Note: For more details on how to investigate unproved checks, see the [SPARK User's Guide³⁸](#).

27.5 Code Examples / Pitfalls

We end with some code examples and pitfalls.

³⁸ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_investigate_unproved_checks.html

27.5.1 Example #1

The package `Lists` defines a linked-list data structure. We call `Link(I,J)` to make a link from index `I` to index `J` and call `Goes_To(I,J)` to determine if we've created a link from index `I` to index `J`. The postcondition of `Link` uses `Goes_To` to state that there must be a link between its arguments once `Link` completes.

Listing 22: lists.ads

```

1  package Lists with SPARK_Mode is
2
3      type Index is new Integer;
4
5      function Goes_To (I, J : Index) return Boolean;
6
7      procedure Link (I, J : Index) with Post => Goes_To (I, J);
8
9  private
10
11     type Cell (Is_Set : Boolean := True) is record
12         case Is_Set is
13             when True =>
14                 Next : Index;
15             when False =>
16                 null;
17         end case;
18     end record;
19
20     type Cell_Array is array (Index) of Cell;
21
22     Memory : Cell_Array;
23
24 end Lists;

```

Listing 23: lists.adb

```

1  package body Lists with SPARK_Mode is
2
3      function Goes_To (I, J : Index) return Boolean is
4          begin
5              if Memory (I).Is_Set then
6                  return Memory (I).Next = J;
7              end if;
8              return False;
9          end Goes_To;
10
11      procedure Link (I, J : Index) is
12          begin
13              Memory (I) := (Is_Set => True, Next => J);
14          end Link;
15
16 end Lists;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
lists.ads:7:47: medium: postcondition might fail [possible fix: you should
  ↵ consider adding a postcondition to function Goes_To or turning it into an
  ↵ expression function]
gnatprove: unproved check messages considered as errors

```

This example is correct, but can't be verified by GNATprove. This is because `Goes_To` itself has no postcondition, so nothing is known about its result.

27.5.2 Example #2

We now redefine `Goes_To` as an expression function.

Listing 24: lists.ads

```

1 package Lists with SPARK_Mode is
2
3     type Index is new Integer;
4
5     function Goes_To (I, J : Index) return Boolean;
6
7     procedure Link (I, J : Index) with Post => Goes_To (I, J);
8
9     private
10
11     type Cell (Is_Set : Boolean := True) is record
12         case Is_Set is
13             when True =>
14                 Next : Index;
15             when False =>
16                 null;
17             end case;
18         end record;
19
20     type Cell_Array is array (Index) of Cell;
21
22     Memory : Cell_Array;
23
24     function Goes_To (I, J : Index) return Boolean is
25         (Memory (I).Is_Set and then Memory (I).Next = J);
26
27 end Lists;

```

Listing 25: lists.adb

```

1 package body Lists with SPARK_Mode is
2
3     procedure Link (I, J : Index) is
4     begin
5         Memory (I) := (Is_Set => True, Next => J);
6     end Link;
7
8 end Lists;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
lists.adb:5:18: info: discriminant check proved
lists.ads:7:47: info: postcondition proved
lists.ads:25:44: info: discriminant check proved

```

GNATprove can fully prove this version: `Goes_To` is an expression function, so its body is available for proof (specifically, for creating the postcondition needed for the proof).

27.5.3 Example #3

The package `Stacks` defines an abstract stack type with a `Push` procedure that adds an element at the top of the stack and a function `Peek` that returns the content of the element at the top of the stack (without removing it).

Listing 26: `stacks.ads`

```

1  package Stacks with SPARK_Mode is
2
3      type Stack is private;
4
5      function Peek (S : Stack) return Natural;
6      procedure Push (S : in out Stack; E : Natural) with
7          Post => Peek (S) = E;
8
9  private
10
11     Max : constant := 10;
12
13     type Stack_Array is array (1 .. Max) of Natural;
14
15     type Stack is record
16         Top      : Positive;
17         Content : Stack_Array;
18     end record;
19
20     function Peek (S : Stack) return Natural is
21         (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
22
23 end Stacks;
```

Listing 27: `stacks.adb`

```

1  package body Stacks with SPARK_Mode is
2
3      procedure Push (S : in out Stack; E : Natural) is
4      begin
5          if S.Top >= Max then
6              return;
7          end if;
8
9          S.Top := S.Top + 1;
10         S.Content (S.Top) := E;
11     end Push;
12
13 end Stacks;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stacks.ads:7:14: medium: postcondition might fail
gnatprove: unproved check messages considered as errors
```

This example isn't correct. The postcondition of `Push` is only satisfied if the stack isn't full when we call `Push`.

27.5.4 Example #4

We now change the behavior of Push so it raises an exception when the stack is full instead of returning.

Listing 28: stacks.ads

```

1 package Stacks with SPARK_Mode is
2
3   type Stack is private;
4
5   Is_Full_E : exception;
6
7   function Peek (S : Stack) return Natural;
8   procedure Push (S : in out Stack; E : Natural) with
9     Post => Peek (S) = E;
10
11  private
12
13    Max : constant := 10;
14
15  type Stack_Array is array (1 .. Max) of Natural;
16
17  type Stack is record
18    Top      : Positive;
19    Content : Stack_Array;
20  end record;
21
22  function Peek (S : Stack) return Natural is
23    (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
24
25 end Stacks;

```

Listing 29: stacks.adb

```

1 package body Stacks with SPARK_Mode is
2
3   procedure Push (S : in out Stack; E : Natural) is
4 begin
5   if S.Top >= Max then
6     raise Is_Full_E;
7   end if;
8
9   S.Top := S.Top + 1;
10  S.Content (S.Top) := E;
11
12 end Push;
13
end Stacks;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stacks.adb:6:10: medium: exception might be raised
gnatprove: unproved check messages considered as errors

```

The postcondition of Push is now proved because GNATprove only considers execution paths leading to normal termination. But it issues a message warning that exception Is_Full_E may be raised at runtime.

27.5.5 Example #5

Let's add a precondition to Push stating that the stack shouldn't be full.

Listing 30: stacks.ads

```

1 package Stacks with SPARK_Mode is
2
3   type Stack is private;
4
5   Is_Full_E : exception;
6
7   function Peek (S : Stack) return Natural;
8   function Is_Full (S : Stack) return Boolean;
9   procedure Push (S : in out Stack; E : Natural) with
10    Pre => not Is_Full (S),
11    Post => Peek (S) = E;
12
13 private
14
15   Max : constant := 10;
16
17   type Stack_Array is array (1 .. Max) of Natural;
18
19   type Stack is record
20     Top      : Positive;
21     Content : Stack_Array;
22   end record;
23
24   function Peek (S : Stack) return Natural is
25     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
26   function Is_Full (S : Stack) return Boolean is (S.Top >= Max);
27
28 end Stacks;

```

Listing 31: stacks.adb

```

1 package body Stacks with SPARK_Mode is
2
3   procedure Push (S : in out Stack; E : Natural) is
4 begin
5   if S.Top >= Max then
6     raise Is_Full_E;
7   end if;
8   S.Top := S.Top + 1;
9   S.Content (S.Top) := E;
10  end Push;
11
12 end Stacks;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stacks.adb:6:10: info: raise statement or expression proved unreachable
stacks.adb:8:22: info: overflow check proved
stacks.adb:9:19: info: index check proved
stacks.ads:11:14: info: postcondition proved
stacks.ads:25:52: info: index check proved

```

This example is correct. With the addition of the precondition, GNATprove can now verify that Is_Full_E can never be raised at runtime.

27.5.6 Example #6

The package `Memories` defines a type `Chunk` that models chunks of memory. Each element of the array, represented by its index, corresponds to one data element. The procedure `Read_Record` reads two pieces of data starting at index `From` out of the chunk represented by the value of `Memory`.

Listing 32: `memories.ads`

```

1  package Memories is
2
3      type Chunk is array (Integer range <>) of Integer
4          with Predicate => Chunk'Length >= 10;
5
6      function Is_Too_Coarse (V : Integer) return Boolean;
7
8      procedure Treat_Value (V : out Integer);
9
10 end Memories;
```

Listing 33: `read_record.adb`

```

1  with Memories; use Memories;
2
3  procedure Read_Record (Memory : Chunk; From : Integer)
4      with SPARK_Mode => On,
5          Pre => From in Memory'First .. Memory'Last - 2
6  is
7      function Read_One (First : Integer; Offset : Integer) return Integer
8          with Pre => First + Offset in Memory'Range
9      is
10         Value : Integer := Memory (First + Offset);
11     begin
12         if Is_Too_Coarse (Value) then
13             Treat_Value (Value);
14         end if;
15         return Value;
16     end Read_One;
17
18     Data1, Data2 : Integer;
19
20 begin
21     Data1 := Read_One (From, 1);
22     Data2 := Read_One (From, 2);
23 end Read_Record;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
read_record.adb:8:24: medium: overflow check might fail, cannot prove lower bound
  ↵for First + Offset [reason for check: result of addition must fit in a 32-bits
  ↵machine integer] [possible fix: use pragma Overflow_Mode or switch -gnat013 or
  ↵unit Ada.Numerics.Big_Numerics.Big_Integer]
read_record.adb:18:04: warning: variable "Data1" is assigned but never read [-
  ↵gnatwm]
read_record.adb:18:11: warning: variable "Data2" is assigned but never read [-
  ↵gnatwm]
read_record.adb:22:04: warning: possibly useless assignment to "Data2", value
  ↵might not be referenced [-gnatwm]
gnatprove: unproved check messages considered as errors
```

This example is correct, but it can't be verified by GNATprove, which analyses Read_One on its own and notices that an overflow may occur in its precondition in certain contexts.

27.5.7 Example #7

Let's rewrite the precondition of Read_One to avoid any possible overflow.

Listing 34: memories.ads

```

1 package Memories is
2
3     type Chunk is array (Integer range <>) of Integer
4         with Predicate => Chunk'Length >= 10;
5
6     function Is_Too_Coarse (V : Integer) return Boolean;
7
8     procedure Treat_Value (V : out Integer);
9
10    end Memories;
```

Listing 35: read_record.adb

```

1 with Memories; use Memories;
2
3 procedure Read_Record (Memory : Chunk; From : Integer)
4     with SPARK_Mode => On,
5          Pre => From in Memory'First .. Memory'Last - 2
6 is
7     function Read_One (First : Integer; Offset : Integer) return Integer
8         with Pre => First >= Memory'First
9             and then Offset in 0 .. Memory'Last - First
10    is
11        Value : Integer := Memory (First + Offset);
12    begin
13        if Is_Too_Coarse (Value) then
14            Treat_Value (Value);
15        end if;
16        return Value;
17    end Read_One;
18
19    Data1, Data2 : Integer;
20
21 begin
22    Data1 := Read_One (From, 1);
23    Data2 := Read_One (From, 2);
24 end Read_Record;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
read_record.adb:9:49: medium: overflow check might fail, cannot prove lower bound
↳ for Memory'Last - First [reason for check: result of subtraction must fit in a
↳ 32-bits machine integer] [possible fix: use pragma Overflow_Mode or switch -
↳ gnato13 or unit Ada.Numerics.Big_Numerics.Big_Integers]
read_record.adb:19:04: warning: variable "Data1" is assigned but never read [-
↳ gnatwm]
read_record.adb:19:11: warning: variable "Data2" is assigned but never read [-
↳ gnatwm]
read_record.adb:23:04: warning: possibly useless assignment to "Data2", value
↳ (continues on next page)
```

(continued from previous page)

```
→might not be referenced [-gnatwm]
gnatprove: unproved check messages considered as errors
```

This example is also not correct: unfortunately, our attempt to correct `Read_One`'s precondition failed. For example, an overflow will occur at runtime if `First` is `Integer'Last` and `Memory'Last` is negative. This is possible here because type `Chunk` uses `Integer` as base index type instead of `Natural` or `Positive`.

27.5.8 Example #8

Let's completely remove the precondition of `Read_One`.

Listing 36: memories.ads

```
1 package Memories is
2
3     type Chunk is array (Integer range <>) of Integer
4         with Predicate => Chunk'Length >= 10;
5
6     function Is_Too_Coarse (V : Integer) return Boolean;
7
8     procedure Treat_Value (V : out Integer);
9
10    end Memories;
```

Listing 37: read_record.adb

```
1 with Memories; use Memories;
2
3 procedure Read_Record (Memory : Chunk; From : Integer)
4     with SPARK_Mode => On,
5          Pre => From in Memory'First .. Memory'Last - 2
6 is
7     function Read_One (First : Integer; Offset : Integer) return Integer is
8         Value : Integer := Memory (First + Offset);
9     begin
10        if Is_Too_Coarse (Value) then
11            Treat_Value (Value);
12        end if;
13        return Value;
14    end Read_One;
15
16    Data1, Data2 : Integer;
17
18 begin
19    Data1 := Read_One (From, 1);
20    Data2 := Read_One (From, 2);
21 end Read_Record;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
read_record.adb:5:51: info: overflow check proved
read_record.adb:8:40: info: overflow check proved, in call inlined at read_record.
→adb:19
read_record.adb:8:40: info: index check proved, in call inlined at read_record.
→adb:19
read_record.adb:8:40: info: overflow check proved, in call inlined at read_record.
(continues on next page)
```

(continued from previous page)

```

adb:20
read_record.adb:8:40: info: index check proved, in call inlined at read_record.
adb:20
read_record.adb:16:04: warning: variable "Data1" is not referenced [-gnatwu]
read_record.adb:16:11: warning: variable "Data2" is not referenced [-gnatwu]

```

This example is correct and fully proved. We could have fixed the contract of `Read_One` to correctly handle both positive and negative values of `Memory'Last`, but we found it simpler to let the function be inlined for proof by removing its precondition.

27.5.9 Example #9

The procedure `Compute` performs various computations on its argument. The computation performed depends on its input range and is reflected in its contract, which we express using a `Contract_Cases` aspect.

Listing 38: `compute.adb`

```

1  procedure Compute (X : in out Integer) with
2      Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
3                             (X in 0 .. 199) => X = X'Old + 1,
4                             (X in -199 .. 0)    => X = X'Old - 1,
5                             X >= 200           => X = 200,
6                             others            => X = -200)
7
8 is
9 begin
10    if X in -100 .. 100 then
11        X := X * 2;
12    elsif X in 0 .. 199 then
13        X := X + 1;
14    elsif X in -199 .. 0 then
15        X := X - 1;
16    elsif X >= 200 then
17        X := 200;
18    else
19        X := -200;
20    end if;
21 end Compute;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
compute.adb:2:03: medium: contract cases might not be disjoint
compute.adb:3:41: medium: contract case might fail
compute.adb:4:41: medium: contract case might fail
gnatprove: unproved check messages considered as errors

```

This example isn't correct. We duplicated the content of `Compute`'s body in its contract. This is incorrect because the semantics of `Contract_Cases` require disjoint cases, just like a `case` statement. The counterexample returned by GNATprove shows that $X = 0$ is covered by two different case-guards (the first and the second).

27.5.10 Example #10

Let's rewrite the contract of Compute to avoid overlapping cases.

Listing 39: compute.adb

```

1  procedure Compute (X : in out Integer) with
2    Contract_Cases => ((X in 0 .. 199) => X >= X'Old,
3                          (X in -199 .. -1) => X <= X'Old,
4                          X >= 200           => X = 200,
5                          X < -200          => X = -200)
6  is
7    begin
8      if X in -100 .. 100 then
9          X := X * 2;
10     elsif X in 0 .. 199 then
11         X := X + 1;
12     elsif X in -199 .. 0 then
13         X := X - 1;
14     elsif X >= 200 then
15         X := 200;
16     else
17         X := -200;
18     end if;
19  end Compute;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
compute.adb:2:03: medium: contract cases might not be complete
gnatprove: unproved check messages considered as errors
```

This example is still not correct. GNATprove can successfully prove the different cases are disjoint and also successfully verify each case individually. This isn't enough, though: a Contract_Cases must cover all cases. Here, we forgot the value -200, which is what GNATprove reports in its counterexample.

STATE ABSTRACTION

Abstraction is a key concept in programming that can drastically simplify both the implementation and maintenance of code. It's particularly well suited to SPARK and its modular analysis. This section explains what state abstraction is and how you use it in SPARK. We explain how it impacts GNATprove's analysis both in terms of information flow and proof of program properties.

State abstraction allows us to:

- express dependencies that wouldn't otherwise be expressible because some data that's read or written isn't visible at the point where a subprogram is declared — examples are dependencies on data, for which we use the Global contract, and on flow, for which we use the Depends contract.
- reduce the number of variables that need to be considered in flow analysis and proof, a reduction which may be critical in order to scale the analysis to programs with thousands of global variables.

28.1 What's an Abstraction?

Abstraction is an important part of programming language design. It provides two views of the same object: an abstract one and a refined one. The abstract one — usually called *specification* — describes what the object does in a coarse way. A subprogram's specification usually describes how it should be called (e.g., parameter information such as how many and of what types) as well as what it does (e.g., returns a result or modifies one or more of its parameters).

Contract-based programming, as supported in Ada, allows contracts to be added to a subprogram's specification. You use contracts to describe the subprogram's behavior in a more fine-grained manner, but all the details of how the subprogram actually works are left to its refined view, its implementation.

Take a look at the example code shown below.

Listing 1: increase.ads

```
1 procedure Increase (X : in out Integer) with
2   Global => null,
3   Pre    => X <= 100,
4   Post   => X'Old < X;
```

Listing 2: increase.adb

```
1 procedure Increase (X : in out Integer) is
2 begin
3   X := X + 1;
4 end Increase;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
increase.adb:3:11: info: overflow check proved
increase.ads:2:03: info: data dependencies proved
increase.ads:4:13: info: postcondition proved
```

We've written a specification of the subprogram Increase to say that it's called with a single argument, a variable of type **Integer** whose initial value is less than 100. Our contract says that the only effect of the subprogram is to increase the value of its argument.

28.2 Why is Abstraction Useful?

A good abstraction of a subprogram's implementation is one whose specification precisely and completely summarizes what its callers can rely on. In other words, a caller of that subprogram shouldn't rely on any behavior of its implementation if that behavior isn't documented in its specification.

For example, callers of the subprogram Increase can assume that it always strictly increases the value of its argument. In the code snippet shown below, this means the loop must terminate.

Listing 3: increase.ads

```
1 procedure Increase (X : in out Integer) with
2   Global => null,
3   Pre    => X <= 100,
4   Post   => X'Old < X;
```

Listing 4: client.adb

```
1 with Increase;
2 procedure Client is
3   X : Integer := 0;
4 begin
5   while X <= 100 loop      -- The loop will terminate
6     Increase (X);          -- Increase can be called safely
7   end loop;
8   pragma Assert (X = 101); -- Will this hold?
9 end Client;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
client.adb:8:19: medium: assertion might fail
gnatprove: unproved check messages considered as errors
```

Callers can also assume that the implementation of Increase won't cause any runtime errors when called in the loop. On the other hand, nothing in the specification guarantees that the assertion shown above is correct: it may fail if Increase's implementation is changed.

If you follow this basic principle, abstraction can bring you significant benefits. It simplifies both your program's implementation and verification. It also makes maintenance and code reuse much easier since changes to the implementation of an object shouldn't affect the code using this object. Your goal in using it is that it should be enough to understand the specification of an object in order to use that object, since understanding the specification is usually much simpler than understanding the implementation.

GNATprove relies on the abstraction defined by subprogram contracts and therefore doesn't prove the assertion after the loop in Client above.

28.3 Abstraction of a Package's State

Subprograms aren't the only objects that benefit from abstraction. The state of a package — the set of persistent variables defined in it — can also be hidden from external users. You achieve this form of abstraction — called *state abstraction* — by defining variables in the body or private part of a package so they can only be accessed through subprogram calls. For example, our Stack package shown below provides an abstraction for a Stack object which can only be modified using the Pop and Push procedures.

```
package Stack is
  procedure Pop  (E : out Element);
  procedure Push (E : in  Element);
end Stack;

package body Stack is
  Content : Element_Array (1 .. Max);
  Top     : Natural;
  ...
end Stack;
```

The fact that we implemented it using an array is irrelevant to the caller. We could change that without impacting our callers' code.

28.4 Declaring a State Abstraction

Hidden state influences a program's behavior, so SPARK allows that state to be declared. You can use the `Abstract_State` aspect, an abstraction that names a state, to do this, but you aren't required to use it even for a package with hidden state. You can use several state abstractions to declare the hidden state of a single package or you can use it for a package with no hidden state at all. However, since SPARK doesn't allow aliasing, different state abstractions must always refer to disjoint sets of variables. A state abstraction isn't a variable: it doesn't have a type and can't be used inside expressions, either those in bodies or contracts.

As an example of the use of this aspect, we can optionally define a state abstraction for the entire hidden state of the Stack package like this:

```
package Stack with
  Abstract_State => The_Stack
is
  ...
```

Alternatively, we can define a state abstraction for each hidden variable:

```
package Stack with
  Abstract_State => (Top_State, Content_State)
is
  ...
```

Remember: a state abstraction isn't a variable (it has no type) and can't be used inside expressions. For example:

```
pragma Assert (Stack.Top_State = ...);
-- compilation error: Top_State is not a variable
```

28.5 Refining an Abstract State

Once you've declared an abstract state in a package, you must refine it into its constituents using a Refined_State aspect. You must place the Refined_State aspect on the package body even if the package wouldn't otherwise have required a body. For each state abstraction you've declared for the package, you list the set of variables represented by that state abstraction in its refined state.

If you specify an abstract state for a package, it must be complete, meaning you must have listed every hidden variable as part of some state abstraction. For example, we must add a Refined_State aspect on our Stack package's body linking the state abstraction (The_Stack) to the entire hidden state of the package, which consists of both Content and Top.

Listing 5: stack.ads

```
1 package Stack with
2   Abstract_State => The_Stack
3 is
4   type Element is new Integer;
5
6   procedure Pop (E : out Element);
7   procedure Push (E : Element);
8
9 end Stack;
```

Listing 6: stack.adb

```
1 package body Stack with
2   Refined_State => (The_Stack => (Content, Top))
3 is
4   Max : constant := 100;
5
6   type Element_Array is array (1 .. Max) of Element;
7
8   Content : Element_Array := (others => 0);
9   Top      : Natural range 0 .. Max := 0;
10  -- Both Content and Top must be listed in the list of
11  -- constituents of The_Stack
12
13  procedure Pop (E : out Element) is
14 begin
15   E := Content (Top);
16   Top := Top - 1;
17 end Pop;
18
19  procedure Push (E : Element) is
20 begin
21   Top      := Top + 1;
22   Content (Top) := E;
23 end Push;
24
25 end Stack;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
stack.ads:6:20: info: initialization of "E" proved
```

28.6 Representing Private Variables

You can refine state abstractions in the package body, where all the variables are visible. When only the package's specification is available, you need a way to specify which state abstraction each private variable belongs to. You do this by adding the `Part_Of` aspect to the variable's declaration.

`Part_Of` annotations are mandatory: if you gave a package an abstract state annotation, you must link all the hidden variables defined in its private part to a state abstraction. For example:

Listing 7: stack.ads

```

1  package Stack with
2    Abstract_State => The_Stack
3  is
4    type Element is new Integer;
5
6    procedure Pop  (E : out Element);
7    procedure Push (E : Element);
8
9  private
10   Max : constant := 100;
11
12  type Element_Array is array (1 .. Max) of Element;
13
14  Content : Element_Array           with Part_Of => The_Stack;
15  Top      : Natural range 0 .. Max with Part_Of => The_Stack;
16
17
18 end Stack;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
```

Since we chose to define `Content` and `Top` in `Stack`'s private part instead of its body, we had to add a `Part_Of` aspect to both of their declarations, associating them with the state abstraction `The_Stack`, even though it's the only state abstraction. However, we still need to list them in the `Refined_State` aspect in `Stack`'s body.

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
```

28.7 Additional State

28.7.1 Nested Packages

So far, we've only discussed hidden variables. But variables aren't the only component of a package's state. If a package P contains a nested package, the nested package's state is also part of P's state. If the nested package is hidden, its state is part of P's hidden state and must be listed in P's state refinement.

We see this in the example below, where the package `Hidden_Nested`'s hidden state is part of P's hidden state.

Listing 8: p.ads

```

1  package P with
2    Abstract_State => State
3  is
4    package Visible_Nested with
5      Abstract_State => Visible_State
6    is
7      procedure Get (E : out Integer);
8    end Visible_Nested;
9  end P;

```

Listing 9: p.adb

```

1  package body P with
2    Refined_State => (State => Hidden_Nested.Hidden_State)
3  is
4    package Hidden_Nested with
5      Abstract_State => Hidden_State,
6      Initializes     => Hidden_State
7    is
8      function Get return Integer;
9    end Hidden_Nested;

10   package body Hidden_Nested with
11     Refined_State => (Hidden_State => Cnt)
12   is
13     Cnt : Integer := 0;

14     function Get return Integer is (Cnt);
15   end Hidden_Nested;

16   package body Visible_Nested with
17     Refined_State => (Visible_State => Checked)
18   is
19     Checked : Boolean := False;

20     procedure Get (E : out Integer) is
21     begin
22       Checked := True;
23       E := Hidden_Nested.Get;
24     end Get;
25   end Visible_Nested;
26 end P;

```

Prover output

Phase 1 of 2: generation of Global contracts ...

(continues on next page)

(continued from previous page)

```
Phase 2 of 2: analysis of data and information flow ...
p.adb:6:07: info: flow dependencies proved
p.adb:14:07: warning: "Cnt" is not modified, could be declared constant [-gnatwk]
p.ads:7:22: info: initialization of "E" proved
```

Any visible state of `Hidden_Nested` would also have been part of P's hidden state. However, if P contains a visible nested package, that nested package's state isn't part of P's hidden state. Instead, you should declare that package's hidden state in a separate state abstraction on its own declaration, like we did above for `Visible_Nested`.

28.7.2 Constants that Depend on Variables

Some constants are also possible components of a state abstraction. These are constants whose value depends either on a variable or a subprogram parameter. They're handled as variables during flow analysis because they participate in the flow of information between variables throughout the program. Therefore, GNATprove considers these constants to be part of a package's state just like it does for variables.

If you've specified a state abstraction for a package, you must list such hidden constants declared in that package in the state abstraction refinement. However, constants that don't depend on variables don't participate in the flow of information and must not appear in a state refinement.

Let's look at this example.

Listing 10: stack.ads

```
1 package Stack with
2   Abstract_State => The_Stack
3 is
4   type Element is new Integer;
5
6   procedure Pop (E : out Element);
7   procedure Push (E : Element);
8 end Stack;
```

Listing 11: configuration.ads

```
1 package Configuration with
2   Initializes => External_Variable
3 is
4   External_Variable : Positive with Volatile;
5 end Configuration;
```

Listing 12: stack.adb

```
1 with Configuration;
2 pragma Elaborate (Configuration);
3
4 package body Stack with
5   Refined_State => (The_Stack => (Content, Top, Max))
6   -- Max has variable inputs. It must appear as a
7   -- constituent of The_Stack
8 is
9   Max : constant Positive := Configuration.External_Variable;
10
11  type Element_Array is array (1 .. Max) of Element;
```

(continues on next page)

(continued from previous page)

```

13 Content : Element_Array := (others => 0);
14 Top      : Natural range 0 .. Max := 0;
15
16 procedure Pop (E : out Element) is
17 begin
18     E   := Content (Top);
19     Top := Top - 1;
20 end Pop;
21
22 procedure Push (E : Element) is
23 begin
24     Top          := Top + 1;
25     Content (Top) := E;
26 end Push;
27
28 end Stack;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
stack.ads:6:20: info: initialization of "E" proved
configuration.ads:2:03: info: flow dependencies proved

```

Here, Max — the maximum number of elements that can be stored in the stack — is initialized from a variable in an external package. Because of this, we must include Max as part of the state abstraction The_Stack.

Note: For more details on state abstractions, see the SPARK User's Guide³⁹.

28.8 Subprogram Contracts

28.8.1 Global and Depends

Hidden variables can only be accessed through subprogram calls, so you document how state abstractions are modified during the program's execution via the contracts of those subprograms. You use Global and Depends contracts to specify which of the state abstractions are used by a subprogram and how values flow through the different variables. The Global and Depends contracts that you write when referring to state abstractions are often less precise than contracts referring to visible variables since the possibly different dependencies of the hidden variables contained within a state abstraction are collapsed into a single dependency.

Let's add Global and Depends contracts to the Pop procedure in our stack.

Listing 13: stack.ads

```

1 package Stack with
2     Abstract_State => (Top_State, Content_State)
3 is
4     type Element is new Integer;
5
6     procedure Pop (E : out Element) with

```

(continues on next page)

³⁹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/package_contracts.html#state-abstraction

(continued from previous page)

```

7   Global  => (Input  => Content_State,
8     In_Out => Top_State),
9   Depends => (Top_State => Top_State,
10    E       => (Content_State, Top_State));
11
12 end Stack;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

```

In this example, the Pop procedure only modifies the value of the hidden variable Top, while Content is unchanged. By using distinct state abstractions for the two variables, we're able to preserve this semantic in the contract.

Let's contrast this example with a different representation of Global and Depends contracts, this time using a single abstract state.

Listing 14: stack.ads

```

1 package Stack with
2   Abstract_State => The_Stack
3 is
4   type Element is new Integer;
5
6   procedure Pop  (E : out Element) with
7     Global  => (In_Out => The_Stack),
8     Depends => ((The_Stack, E) => The_Stack);
9
10 end Stack;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

```

Here, Top_State and Content_State are merged into a single state abstraction, The_Stack. By doing so, we've hidden the fact that Content isn't modified (though we're still showing that Top may be modified). This loss in precision is reasonable here, since it's the whole point of the abstraction. However, you must be careful not to aggregate unrelated hidden state because this risks their annotations becoming meaningless.

Even though imprecise contracts that consider state abstractions as a whole are perfectly reasonable for users of a package, you should write Global and Depends contracts that are as precise as possible within the package body. To allow this, SPARK introduces the notion of *refined contracts*, which are precise contracts specified on the bodies of subprograms where state refinements are visible. These contracts are the same as normal Global and Depends contracts except they refer directly to the hidden state of the package.

When a subprogram is called inside the package body, you should write refined contracts instead of the general ones so that the verification can be as precise as possible. However, refined Global and Depends are optional: if you don't specify them, GNATprove will compute them to check the package's implementation.

For our Stack example, we could add refined contracts as shown below.

Listing 15: stack.ads

```

1 package Stack with
2   Abstract_State => The_Stack

```

(continues on next page)

(continued from previous page)

```

3  is
4    type Element is new Integer;
5
6    procedure Pop (E : out Element) with
7      Global => (In_Out => The_Stack),
8      Depends => ((The_Stack, E) => The_Stack);
9
10   procedure Push (E : Element) with
11     Global => (In_Out => The_Stack),
12     Depends => (The_Stack => (The_Stack, E));
13
14 end Stack;

```

Listing 16: stack.adb

```

1 package body Stack with
2   Refined_State => (The_Stack => (Content, Top))
3 is
4   Max : constant := 100;
5
6   type Element_Array is array (1 .. Max) of Element;
7
8   Content : Element_Array := (others => 0);
9   Top      : Natural range 0 .. Max := 0;
10
11  procedure Pop (E : out Element) with
12    Refined_Global => (Input => Content,
13                          In_Out => Top),
14    Refined_Depends => (Top => Top,
15                          E => (Content, Top))
16  is
17  begin
18    E := Content (Top);
19    Top := Top - 1;
20  end Pop;
21
22  procedure Push (E : Element) with
23    Refined_Global => (In_Out => (Content, Top)),
24    Refined_Depends => (Content =>+ (Content, Top, E),
25                          Top => Top) is
26  begin
27    Top := Top + 1;
28    Content (Top) := E;
29  end Push;
30
31 end Stack;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...

```

28.8.2 Preconditions and Postconditions

We mostly express functional properties of subprograms using preconditions and postconditions. These are standard Boolean expressions, so they can't directly refer to state abstractions. To work around this restriction, we can define functions to query the value of hidden variables. We then use these functions in place of the state abstraction in the contract of other subprograms.

For example, we can query the state of the stack with functions `Is_Empty` and `Is_Full` and call these in the contracts of procedures `Pop` and `Push`:

Listing 17: stack.ads

```

1 package Stack is
2   type Element is new Integer;
3
4   function Is_Empty return Boolean;
5   function Is_Full return Boolean;
6
7   procedure Pop (E : out Element) with
8     Pre => not Is_Empty,
9     Post => not Is_Full;
10
11  procedure Push (E : Element) with
12    Pre => not Is_Full,
13    Post => not Is_Empty;
14
15 end Stack;
```

Listing 18: stack.adb

```

1 package body Stack is
2
3   Max : constant := 100;
4
5   type Element_Array is array (1 .. Max) of Element;
6
7   Content : Element_Array := (others => 0);
8   Top      : Natural range 0 .. Max := 0;
9
10  function Is_Empty return Boolean is (Top = 0);
11  function Is_Full  return Boolean is (Top = Max);
12
13  procedure Pop (E : out Element) is
14    begin
15      E := Content (Top);
16      Top := Top - 1;
17    end Pop;
18
19  procedure Push (E : Element) is
20    begin
21      Top      := Top + 1;
22      Content (Top) := E;
23    end Push;
24
25 end Stack;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.adb:15:23: info: index check proved
```

(continues on next page)

(continued from previous page)

```
stack.adb:16:18: info: range check proved
stack.adb:21:28: info: range check proved
stack.adb:22:16: info: index check proved
stack.ads:7:19: info: initialization of "E" proved
stack.ads:9:14: info: postcondition proved
stack.ads:13:14: info: postcondition proved
```

Just like we saw for Global and Depends contracts, you may often find it useful to have a more precise view of functional contracts in the context where the hidden variables are visible. You do this using expression functions in the same way we did for the functions Is_Empty and Is_Full above. As expression function, bodies act as contracts for GNATprove, so they automatically give a more precise version of the contracts when their implementation is visible.

You may often need a more constraining contract to verify the package's implementation but want to be less strict outside the abstraction. You do this using the Refined_Post aspect. This aspect, when placed on a subprogram's body, provides stronger guarantees to internal callers of a subprogram. If you provide one, the refined postcondition must imply the subprogram's postcondition. This is checked by GNATprove, which reports a failing postcondition if the refined postcondition is too weak, even if it's actually implied by the subprogram's body. SPARK doesn't perform a similar verification for normal preconditions.

For example, we can refine the postconditions in the bodies of Pop and Push to be more detailed than what we wrote for them in their specification.

Listing 19: stack.ads

```
1 package Stack is
2   type Element is new Integer;
3
4   function Is_Empty return Boolean;
5   function Is_Full return Boolean;
6
7   procedure Pop (E : out Element) with
8     Pre => not Is_Empty,
9     Post => not Is_Full;
10
11  procedure Push (E : Element) with
12    Pre => not Is_Full,
13    Post => not Is_Empty;
14
15 end Stack;
```

Listing 20: stack.adb

```
1 package body Stack is
2
3   Max : constant := 100;
4
5   type Element_Array is array (1 .. Max) of Element;
6
7   Content : Element_Array := (others => 0);
8   Top      : Natural range 0 .. Max := 0;
9
10  function Is_Empty return Boolean is (Top = 0);
11  function Is_Full  return Boolean is (Top = Max);
12
13  procedure Pop (E : out Element) with
14    Refined_Post => not Is_Full and E = Content (Top)'Old
15    is
```

(continues on next page)

(continued from previous page)

```

16 begin
17   E := Content (Top);
18   Top := Top - 1;
19 end Pop;

20
21 procedure Push (E : Element) with
22   Refined_Post => not Is_Empty and E = Content (Top)
23 is
24 begin
25   Top          := Top + 1;
26   Content (Top) := E;
27 end Push;

28
29 end Stack;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.adb:14:22: info: refined post proved
stack.adb:14:51: info: index check proved
stack.adb:17:23: info: index check proved
stack.adb:18:18: info: range check proved
stack.adb:22:22: info: refined post proved
stack.adb:22:52: info: index check proved
stack.adb:25:28: info: range check proved
stack.adb:26:16: info: index check proved
stack.ads:7:19: info: initialization of "E" proved
stack.ads:9:14: info: postcondition proved
stack.ads:13:14: info: postcondition proved

```

Note: For more details on refinement in contracts, see the SPARK User's Guide⁴⁰.

28.9 Initialization of Local Variables

As part of flow analysis, GNATprove checks for the proper initialization of variables. Therefore, flow analysis needs to know which variables are initialized during the package's elaboration.

You can use the `Initializes` aspect to specify the set of visible variables and state abstractions that are initialized during the elaboration of a package. An `Initializes` aspect can't refer to a variable that isn't defined in the unit since, in SPARK, a package can only initialize variables declared immediately within the package.

`Initializes` aspects are optional. If you don't supply any, they'll be derived by GNATprove.

For our `Stack` example, we could add an `Initializes` aspect.

Listing 21: stack.ads

```

1 package Stack with
2   Abstract_State => The_Stack,
3   Initializes    => The_Stack
4 is

```

(continues on next page)

⁴⁰ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/subprogram_contracts.html#state-abstraction-and-contracts

(continued from previous page)

```

5   type Element is new Integer;
6
7   procedure Pop (E : out Element);
8
9 end Stack;
```

Listing 22: stack.adb

```

1 package body Stack with
2   Refined_State => (The_Stack => (Content, Top))
3 is
4   Max : constant := 100;
5
6   type Element_Array is array (1 .. Max) of Element;
7
8   Content : Element_Array := (others => 0);
9   Top      : Natural range 0 .. Max := 0;
10
11  procedure Pop (E : out Element) is
12 begin
13   E := Content (Top);
14   Top := Top - 1;
15 end Pop;
16
17 end Stack;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
stack.adb:8:04: warning: "Content" is not modified, could be declared constant [-gnatwk]
stack.ads:3:03: info: flow dependencies proved
stack.ads:7:20: info: initialization of "E" proved
```

Flow analysis also checks for dependencies between variables, so it must be aware of how information flows through the code that performs the initialization of states. We discussed one use of the `Initializes` aspect above. But you also can use it to provide flow information. If the initial value of a variable or state abstraction is dependent on the value of another visible variable or state abstraction from another package, you must list this dependency in the `Initializes` contract. You specify the list of entities on which a variable's initial value depends using an arrow following that variable's name.

Let's look at this example:

Listing 23: q.ads

```

1 package Q is
2   External_Variable : Integer := 2;
3 end Q;
```

Listing 24: p.ads

```

1 with Q;
2 package P with
3   Initializes => (V1, V2 => Q.External_Variable)
4 is
5   V1 : Integer := 0;
6   V2 : Integer := Q.External_Variable;
7 end P;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
p.ads:3:03: info: flow dependencies proved
```

Here we indicated that V2's initial value depends on the value of Q.External_Variable by including that dependency in the Initializes aspect of P. We didn't list any dependency for V1 because its initial value doesn't depend on any external variable. We could also have stated that lack of dependency explicitly by writing V1 => **null**.

GNATprove computes dependencies of initial values if you don't supply an Initializes aspect. However, if you do provide an Initializes aspect for a package, it must be complete: you must list every initialized state of the package, along with all its external dependencies.

Note: For more details on Initializes, see the [SPARK User's Guide⁴¹](#).

28.10 Code Examples / Pitfalls

This section contains some code examples to illustrate potential pitfalls.

28.10.1 Example #1

Package Communication defines a hidden local package, Ring_Buffer, whose capacity is initialized from an external configuration during elaboration.

Listing 25: configuration.ads

```
1 package Configuration is
2   External_Variable : Natural := 1;
3 end Configuration;
```

Listing 26: communication.ads

```
1 with Configuration;
2
3 package Communication with
4   Abstract_State => State,
5   Initializes     => (State => Configuration.External_Variable)
6 is
7   function Get_Capacity return Natural;
8
9 private
10
11   package Ring_Buffer with
12     Initializes => (Capacity => Configuration.External_Variable)
13   is
14     Capacity : constant Natural := Configuration.External_Variable;
15   end Ring_Buffer;
```

(continues on next page)

⁴¹ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/package_contracts.html#package-initialization

(continued from previous page)

```
16
17 end Communication;
```

Listing 27: communication.adb

```
1 package body Communication with
2   Refined_State => (State => Ring_Buffer.Capacity)
3 is
4
5   function Get_Capacity return Natural is
6   begin
7     return Ring_Buffer.Capacity;
8   end Get_Capacity;
9
10 end Communication;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
communication.adb:2:41: error: "Capacity" cannot act as constituent of state "State"
  ↵
communication.adb:2:41: error: missing Part_Of indicator at communication.ads:14
  ↵should specify encapsulator "State"
gnatprove: error during generation of Global contracts
```

This example isn't correct. Capacity is declared in the private part of Communication. Therefore, we should have linked it to State by using the Part_Of aspect in its declaration.

28.10.2 Example #2

Let's add Part_Of to the state of hidden local package Ring_Buffer, but this time we hide variable Capacity inside the private part of Ring_Buffer.

Listing 28: configuration.ads

```
1 package Configuration is
2
3   External_Variable : Natural := 1;
4
5 end Configuration;
```

Listing 29: communication.ads

```
1 with Configuration;
2
3 package Communication with
4   Abstract_State => State
5 is
6   private
7
8     package Ring_Buffer with
9       Abstract_State => (B_State with Part_Of => State),
10      Initializes    => (B_State => Configuration.External_Variable)
11   is
12     function Get_Capacity return Natural;
13   private
14     Capacity : constant Natural := Configuration.External_Variable
15     with Part_Of => B_State;
```

(continues on next page)

(continued from previous page)

```

16  end Ring_Buffer;
17
18 end Communication;
```

Listing 30: communication.adb

```

1 package body Communication with
2   Refined_State => (State => Ring_Buffer.B_State)
3 is
4
5   package body Ring_Buffer with
6     Refined_State => (B_State => Capacity)
7   is
8     function Get_Capacity return Natural is (Capacity);
9   end Ring_Buffer;
10
11 end Communication;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
communication.ads:10:06: info: flow dependencies proved
```

This program is correct and GNATprove is able to verify it.

28.10.3 Example #3

Package Counting defines two counters: Black_Counter and Red_Counter. It provides separate initialization procedures for each, both called from the main procedure.

Listing 31: counting.ads

```

1 package Counting with
2   Abstract_State => State
3 is
4   procedure Reset_Black_Count;
5   procedure Reset_Red_Count;
6 end Counting;
```

Listing 32: counting.adb

```

1 package body Counting with
2   Refined_State => (State => (Black_Counter, Red_Counter))
3 is
4   Black_Counter, Red_Counter : Natural;
5
6   procedure Reset_Black_Count is
7   begin
8     Black_Counter := 0;
9   end Reset_Black_Count;
10
11  procedure Reset_Red_Count is
12  begin
13    Red_Counter := 0;
14  end Reset_Red_Count;
15 end Counting;
```

Listing 33: main.adb

```

1  with Counting; use Counting;
2
3  procedure Main is
4  begin
5      Reset_Black_Count;
6      Reset_Red_Count;
7  end Main;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
main.adb:5:04: medium: "Counting.State" might not be initialized after elaboration
  ↪ of main program "Main"
counting.ads:2:21: warning: no procedure exists that can initialize abstract state
  ↪ "Counting.State"
gnatprove: unproved check messages considered as errors

```

This program doesn't read any uninitialized data, but GNATprove fails to verify that. This is because we provided a state abstraction for package Counting, so flow analysis computes the effects of subprograms in terms of this state abstraction and thus considers State to be an in-out global consisting of both Black_Counter and Red_Counter. So it issues the message requiring that State be initialized after elaboration as well as the warning that no procedure in package Counting can initialize its state.

28.10.4 Example #4

Let's remove the abstract state on package Counting.

Listing 34: counting.ads

```

1  package Counting is
2      procedure Reset_Black_Count;
3      procedure Reset_Red_Count;
4  end Counting;

```

Listing 35: counting.adb

```

1  package body Counting is
2      Black_Counter, Red_Counter : Natural;
3
4      procedure Reset_Black_Count is
5      begin
6          Black_Counter := 0;
7      end Reset_Black_Count;
8
9      procedure Reset_Red_Count is
10     begin
11         Red_Counter := 0;
12     end Reset_Red_Count;
13  end Counting;

```

Listing 36: main.adb

```

1  with Counting; use Counting;
2

```

(continues on next page)

(continued from previous page)

```

3 procedure Main is
4 begin
5   Reset_Black_Count;
6   Reset_Red_Count;
7 end Main;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
counting.adb:2:04: warning: variable "Black_Counter" is assigned but never read [-gnatwm]
counting.adb:2:19: warning: variable "Red_Counter" is assigned but never read [-gnatwm]

```

This example is correct. Because we didn't provide a state abstraction, GNATprove reasons in terms of variables, instead of states, and proves data initialization without any problem.

28.10.5 Example #5

Let's restore the abstract state to package Counting, but this time provide a procedure `Reset_All` that calls the initialization procedures `Reset_Black_Counter` and `Reset_Red_Counter`.

Listing 37: counting.ads

```

1 package Counting with
2   Abstract_State => State
3 is
4   procedure Reset_Black_Count with Global => (In_Out => State);
5   procedure Reset_Red_Count with Global => (In_Out => State);
6   procedure Reset_All      with Global => (Output => State);
7 end Counting;

```

Listing 38: counting.adb

```

1 package body Counting with
2   Refined_State => (State => (Black_Counter, Red_Counter))
3 is
4   Black_Counter, Red_Counter : Natural;
5
6   procedure Reset_Black_Count is
7   begin
8     Black_Counter := 0;
9   end Reset_Black_Count;
10
11  procedure Reset_Red_Count is
12  begin
13    Red_Counter := 0;
14  end Reset_Red_Count;
15
16  procedure Reset_All is
17  begin
18    Reset_Black_Count;
19    Reset_Red_Count;
20  end Reset_All;
21 end Counting;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
counting.ads:4:37: info: data dependencies proved
counting.ads:5:37: info: data dependencies proved
counting.ads:6:14: info: initialization of "Black_Counter" constituent of "State" ↴
    ↴ proved
counting.ads:6:14: info: initialization of "Red_Counter" constituent of "State" ↴
    ↴ proved
counting.ads:6:37: info: data dependencies proved

```

This example is correct. Flow analysis computes refined versions of Global contracts for internal calls and uses these to verify that `Reset_All` indeed properly initializes `State`. The `Refined_Global` and `Global` annotations are not mandatory and can be computed by GNATprove.

28.10.6 Example #6

Let's consider yet another version of our abstract stack unit.

Listing 39: stack.ads

```

1 package Stack with
2     Abstract_State => The_Stack
3 is
4     pragma Unevaluated_Use_of_Old (Allow);
5
6     type Element is new Integer;
7
8     type Element_Array is array (Positive range <>) of Element;
9     Max : constant Natural := 100;
10    subtype Length_Type is Natural range 0 .. Max;
11
12   procedure Push (E : Element) with
13       Post =>
14         not Is_Empty and
15           (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);
16
17   function Peek      return Element with Pre => not Is_Empty;
18   function Is_Full  return Boolean;
19   function Is_Empty return Boolean;
20 end Stack;

```

Listing 40: stack.adb

```

1 package body Stack with
2     Refined_State => (The_Stack => (Top, Content))
3 is
4     Top      : Length_Type := 0;
5     Content : Element_Array (1 .. Max) := (others => 0);
6
7     procedure Push (E : Element) is
8     begin
9         Top          := Top + 1;
10        Content (Top) := E;
11    end Push;
12
13    function Peek      return Element is (Content (Top));
14    function Is_Full  return Boolean is (Top >= Max);
15    function Is_Empty return Boolean is (Top = 0);
16 end Stack;

```

Build output

```
stack.ads:15:39: error: there is no applicable operator "=" for package or
procedure name
gprbuild: *** compilation phase failed
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
stack.ads:15:39: error: there is no applicable operator "=" for package or
procedure name
gnatprove: error during generation of Global contracts
```

This example isn't correct. There's a compilation error in Push's postcondition: The_Stack is a state abstraction, not a variable, and therefore can't be used in an expression.

28.10.7 Example #7

In this version of our abstract stack unit, a copy of the stack is returned by function Get_Stack, which we call in the postcondition of Push to specify that the stack shouldn't be modified if it's full. We also assert that after we push an element on the stack, either the stack is unchanged (if it was already full) or its top element is equal to the element just pushed.

Listing 41: stack.ads

```

1  package Stack with
2    Abstract_State => The_Stack
3  is
4    pragma Unevaluated_Use_of_Old (Allow);
5
6    type Stack_Model is private;
7
8    type Element is new Integer;
9    type Element_Array is array (Positive range <>) of Element;
10   Max : constant Natural := 100;
11   subtype Length_Type is Natural range 0 .. Max;
12
13  function Peek      return Element with Pre => not Is_Empty;
14  function Is_Full   return Boolean;
15  function Is_Empty  return Boolean;
16  function Get_Stack return Stack_Model;
17
18  procedure Push (E : Element) with
19    Post => not Is_Empty and
20    (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
21
22  private
23
24  type Stack_Model is record
25    Top      : Length_Type := 0;
26    Content : Element_Array (1 .. Max) := (others => 0);
27  end record;
28
29 end Stack;
```

Listing 42: stack.adb

```

1  package body Stack with
2    Refined_State => (The_Stack => (Top, Content))
```

(continues on next page)

(continued from previous page)

```

3   is
4     Top      : Length_Type := 0;
5     Content : Element_Array (1 .. Max) := (others => 0);
6
7   procedure Push (E : Element) is
8   begin
9     if Top >= Max then
10       return;
11     end if;
12     Top          := Top + 1;
13     Content (Top) := E;
14   end Push;
15
16   function Peek    return Element is (Content (Top));
17   function Is_Full return Boolean is (Top >= Max);
18   function Is_Empty return Boolean is (Top = 0);
19
20   function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));
21
22 end Stack;

```

Listing 43: use_stack.adb

```

1  with Stack; use Stack;
2
3  procedure Use_Stack (E : Element) with
4    Pre => not Is_Empty
5  is
6    F : Element := Peek;
7  begin
8    Push (E);
9    pragma Assert (Peek = E or Peek = F);
10   end Use_Stack;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
use_stack.adb:6:04: warning: "F" is not modified, could be declared constant [-gnatwk]
use_stack.adb:9:19: medium: assertion might fail [possible fix: precondition of
  ↵subprogram at line 3 should mention E]
gnatprove: unproved check messages considered as errors

```

This program is correct, but GNATprove can't prove the assertion in Use_Stack. Indeed, even if Get_Stack is an expression function, its body isn't visible outside of Stack's body, where it's defined.

28.10.8 Example #8

Let's move the definition of Get_Stack and other expression functions inside the private part of the spec of Stack.

Listing 44: stack.ads

```

1  package Stack with
2    Abstract_State => The_Stack
3  is
4    pragma Unevaluated_Use_of_Old (Allow);

```

(continues on next page)

(continued from previous page)

```

5   type Stack_Model is private;
6
7
8   type Element is new Integer;
9   type Element_Array is array (Positive range <>) of Element;
10  Max : constant Natural := 100;
11  subtype Length_Type is Natural range 0 .. Max;
12
13  function Peek      return Element with Pre => not Is_Empty;
14  function Is_Full    return Boolean;
15  function Is_Empty   return Boolean;
16  function Get_Stack  return Stack_Model;
17
18  procedure Push (E : Element) with
19    Post => not Is_Empty and
20      (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
21
22 private
23
24  Top      : Length_Type          := 0 with Part_0f => The_Stack;
25  Content : Element_Array (1 .. Max) := (others => 0) with
26    Part_0f => The_Stack;
27
28  type Stack_Model is record
29    Top      : Length_Type := 0;
30    Content : Element_Array (1 .. Max) := (others => 0);
31  end record;
32
33  function Peek      return Element      is (Content (Top));
34  function Is_Full   return Boolean     is (Top >= Max);
35  function Is_Empty  return Boolean     is (Top = 0);
36
37  function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));
38
39 end Stack;

```

Listing 45: stack.adb

```

1 package body Stack with
2   Refined_State => (The_Stack => (Top, Content))
3 is
4
5   procedure Push (E : Element) is
6   begin
7     if Top >= Max then
8       return;
9     end if;
10    Top      := Top + 1;
11    Content (Top) := E;
12  end Push;
13
14 end Stack;

```

Listing 46: use_stack.adb

```

1 with Stack; use Stack;
2
3 procedure Use_Stack (E : Element) with
4   Pre => not Is_Empty
5 is

```

(continues on next page)

(continued from previous page)

```

6   F : Element := Peek;
7   begin
8     Push (E);
9     pragma Assert (Peek = E or Peek = F);
10    end Use_Stack;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
use_stack.adb:6:04: warning: "F" is not modified, could be declared constant [-gnatwk]
use_stack.adb:6:19: info: precondition proved
use_stack.adb:9:19: info: precondition proved
use_stack.adb:9:19: info: assertion proved
use_stack.adb:9:31: info: precondition proved
stack.adb:10:30: info: range check proved
stack.adb:11:16: info: index check proved
stack.ads:19:14: info: postcondition proved
stack.ads:20:60: info: precondition proved
stack.ads:33:55: info: index check proved
```

This example is correct. GNATprove can verify the assertion in `Use_Stack` because it has visibility to `Get_Stack`'s body.

28.10.9 Example #9

Package `Data` defines three variables, `Data_1`, `Data_2` and `Data_3`, that are initialized at elaboration (in `Data`'s package body) from an external interface that reads the file system.

Listing 47: `external_interface.ads`

```

1  package External_Interface with
2    Abstract_State => File_System,
3    Initializes      => File_System
4  is
5    type Data_Type_1 is new Integer;
6    type Data_Type_2 is new Integer;
7    type Data_Type_3 is new Integer;
8
9    type Data_Record is record
10      Field_1 : Data_Type_1;
11      Field_2 : Data_Type_2;
12      Field_3 : Data_Type_3;
13    end record;
14
15   procedure Read_Data (File_Name : String; Data : out Data_Record)
16   with Global => File_System;
17 end External_Interface;
```

Listing 48: `data.ads`

```

1  with External_Interface; use External_Interface;
2
3  package Data with
4    Initializes => (Data_1, Data_2, Data_3)
5  is
6    pragma Elaborate_Body;
```

(continues on next page)

(continued from previous page)

```

8  Data_1 : Data_Type_1;
9  Data_2 : Data_Type_2;
10 Data_3 : Data_Type_3;
11
12 end Data;

```

Listing 49: data.adb

```

1  with External_Interface;
2  pragma Elaborate_All (External_Interface);
3
4  package body Data is
5    begin
6      declare
7        Data_Read : Data_Record;
8      begin
9        Read_Data ("data_file_name", Data_Read);
10       Data_1 := Data_Read.Field_1;
11       Data_2 := Data_Read.Field_2;
12       Data_3 := Data_Read.Field_3;
13     end;
14 end Data;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
data.adb:9:07: high: "External_Interface.File_System" must be mentioned as an
  ↪input of the Initializes aspect of "Data" (SPARK RM 7.1.5(11))
gnatprove: unproved check messages considered as errors

```

This example isn't correct. The dependency between Data_1's, Data_2's, and Data_3's initial values and File_System must be listed in Data's Initializes aspect.

28.10.10 Example #10

Let's remove the Initializes contract on package Data.

Listing 50: external_interface.ads

```

1  package External_Interface with
2    Abstract_State => File_System,
3    Initializes     => File_System
4  is
5    type Data_Type_1 is new Integer;
6    type Data_Type_2 is new Integer;
7    type Data_Type_3 is new Integer;
8
9    type Data_Record is record
10      Field_1 : Data_Type_1;
11      Field_2 : Data_Type_2;
12      Field_3 : Data_Type_3;
13    end record;
14
15    procedure Read_Data (File_Name : String; Data : out Data_Record)
16      with Global => File_System;
17 end External_Interface;

```

Listing 51: data.ads

```
1  with External_Interface; use External_Interface;
2
3  package Data is
4      pragma Elaborate_Body;
5
6      Data_1 : Data_Type_1;
7      Data_2 : Data_Type_2;
8      Data_3 : Data_Type_3;
9
10 end Data;
```

Listing 52: data.adb

```
1  with External_Interface;
2  pragma Elaborate_All (External_Interface);
3
4  package body Data is
5    begin
6      declare
7          Data_Read : Data_Record;
8      begin
9          Read_Data ("data_file_name", Data_Read);
10         Data_1 := Data_Read.Field_1;
11         Data_2 := Data_Read.Field_2;
12         Data_3 := Data_Read.Field_3;
13     end;
14 end Data;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
data.adb:7:07: info: initialization of "Data_Read" proved
external_interface.ads:3:03: info: flow dependencies proved
```

This example is correct. Since Data has no Initializes aspect, GNATprove computes the set of variables initialized during its elaboration as well as their dependencies.

PROOF OF FUNCTIONAL CORRECTNESS

This section is dedicated to the functional correctness of programs. It presents advanced proof features that you may need to use for the specification and verification of your program's complex properties.

29.1 Beyond Program Integrity

When we speak about the *correctness* of a program or subprogram, we mean the extent to which it complies with its specification. Functional correctness is specifically concerned with properties that involve the relations between the subprogram's inputs and outputs, as opposed to other properties such as running time or memory consumption.

For functional correctness, we usually specify stronger properties than those required to just prove program integrity. When we're involved in a certification processes, we should derive these properties from the requirements of the system, but, especially in non-certification contexts, they can also come from more informal sources, such as the program's documentation, comments in its code, or test oracles.

For example, if one of our goals is to ensure that no runtime error is raised when using the result of the function Find below, it may be enough to know that the result is either 0 or in the range of A. We can express this as a postcondition of Find.

Listing 1: show_find.ads

```
1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural with
6         Post => Find'Result in 0 | A'Range;
7
8 end Show_Find;
```

Listing 2: show_find.adb

```
1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4         begin
5             for I in A'Range loop
6                 if A (I) = E then
7                     return I;
8                 end if;
9             end loop;
10            return 0;
11        end Find;
```

(continues on next page)

(continued from previous page)

```
12
13 end Show_Find;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:7:20: info: range check proved
show_find.ads:6:14: info: postcondition proved
```

In this case, it's automatically proved by GNATprove.

However, to be sure that Find performs the task we expect, we may want to verify more complex properties of that function. For example, we want to ensure it returns an index of A where E is stored and returns 0 only if E is nowhere in A. Again, we can express this as a postcondition of Find.

Listing 3: show_find.ads

```
1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural with
6         Post =>
7             (if (for all I in A'Range => A (I) /= E)
8                 then Find'Result = 0
9                 else Find'Result in A'Range and then A (Find'Result) = E);
10
11 end Show_Find;
```

Listing 4: show_find.adb

```
1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4         begin
5             for I in A'Range loop
6                 if A (I) = E then
7                     return I;
8                 end if;
9             end loop;
10            return 0;
11        end Find;
12
13 end Show_Find;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.ads:9:14: medium: postcondition might fail, cannot prove Find'Result in A
    'range
gnatprove: unproved check messages considered as errors
```

This time, GNATprove can't prove this postcondition automatically, but we'll see later that we can help GNATprove by providing a loop invariant, which is checked by GNATprove and allows it to automatically prove the postcondition for Find.

Writing at least part of your program's specification in the form of contracts has many advantages. You can execute those contracts during testing, which improves the maintain-

ability of the code by detecting discrepancies between the program and its specification in earlier stages of development. If the contracts are precise enough, you can use them as oracles to decide whether a given test passed or failed. In that case, they can allow you to verify the outputs of specific subprograms while running a larger block of code. This may, in certain contexts, replace the need for you to perform unit testing, instead allowing you to run integration tests with assertions enabled. Finally, if the code is in SPARK, you can also use GNATprove to formally prove these contracts.

The advantage of a formal proof is that it verifies all possible execution paths, something which isn't always possible by running test cases. For example, during testing, the post-condition of the subprogram `Find` shown below is checked dynamically for the set of inputs for which `Find` is called in that test, but just for that set.

Listing 5: show_find.ads

```

1 package Show_Find is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function Find (A : Nat_Array; E : Natural) return Natural with
6     Post =>
7       (if (for all I in A'Range => A (I) /= E)
8        then Find'Result = 0
9        else Find'Result in A'Range and then A (Find'Result) = E);
10
11 end Show_Find;
```

Listing 6: show_find.adb

```

1 package body Show_Find is
2
3   function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5       for I in A'Range loop
6         if A (I) = E then
7           return I;
8         end if;
9       end loop;
10      return 0;
11    end Find;
12
13 end Show_Find;
```

Listing 7: use_find.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Show_Find; use Show_Find;
3
4 procedure Use_Find with
5   SPARK_Mode => Off
6 is
7   Seq : constant Nat_Array (1 .. 3) := (1, 5, 3);
8   Res : Natural;
9 begin
10   Res := Find (Seq, 3);
11   Put_Line ("Found 3 in index #" & Natural'Image (Res) & " of array");
12 end Use_Find;
```

Prover output

Phase 1 of 2: generation of Global contracts ...

(continues on next page)

(continued from previous page)

```
Phase 2 of 2: flow analysis and proof ...
show_find.ads:9:14: medium: postcondition might fail, cannot prove Find'Result in A
  ↵'range
gnatprove: unproved check messages considered as errors
```

Runtime output

```
Found 3 in index # 3 of array
```

However, if `Find` is formally verified, that verification checks its postcondition for all possible inputs. During development, you can attempt such verification earlier than testing since it's performed modularly on a per-subprogram basis. For example, in the code shown above, you can formally verify `Use_Find` even before you write the body for subprogram `Find`.

29.2 Advanced Contracts

Contracts for functional correctness are usually more complex than contracts for program integrity, so they more often require you to use the new forms of expressions introduced by the Ada 2012 standard. In particular, quantified expressions, which allow you to specify properties that must hold for all or for at least one element of a range, come in handy when specifying properties of arrays.

As contracts become more complex, you may find it useful to introduce new abstractions to improve the readability of your contracts. Expression functions are a good means to this end because you can retain their bodies in your package's specification.

Finally, some properties, especially those better described as invariants over data than as properties of subprograms, may be cumbersome to express as subprogram contracts. Type predicates, which must hold for every object of a given type, are usually a better match for this purpose. Here's an example.

Listing 8: `show_sort.ads`

```
1 package Show_Sort is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function Is_Sorted (A : Nat_Array) return Boolean is
6     (for all I in A'Range =>
7      (if I < A'Last then A (I) <= A (I + 1)));
8     -- Returns True if A is sorted in increasing order.
9
10  subtype Sorted_Nat_Array is Nat_Array with
11    Dynamic_Predicate => Is_Sorted (Sorted_Nat_Array);
12    -- Elements of type Sorted_Nat_Array are all sorted.
13
14  Good_Array : Sorted_Nat_Array := (1, 2, 4, 8, 42);
15 end Show_Sort;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_sort.ads:7:32: info: index check proved
show_sort.ads:7:43: info: overflow check proved
show_sort.ads:7:43: info: index check proved
show_sort.ads:14:37: info: range check proved
show_sort.ads:14:37: info: predicate check proved
```

We can use the subtype `Sorted_Nat_Array` as the type of a variable that must remain sorted throughout the program's execution. Specifying that an array is sorted requires a rather complex expression involving quantifiers, so we abstract away this property as an expression function to improve readability. `Is_Sorted`'s body remains in the package's specification and allows users of the package to retain a precise knowledge of its meaning when necessary. (You must use `Nat_Array` as the type of the operand of `Is_Sorted`. If you use `Sorted_Nat_Array`, you'll get infinite recursion at runtime when assertion checks are enabled since that function is called to check all operands of type `Sorted_Nat_Array`.)

29.2.1 Ghost Code

As the properties you need to specify grow more complex, you may have entities that are only needed because they are used in specifications (contracts). You may find it important to ensure that these entities can't affect the behavior of the program or that they're completely removed from production code. This concept, having entities that are only used for specifications, is usually called having *ghost* code and is supported in SPARK by the Ghost aspect.

You can use Ghost aspects to annotate any entity including variables, types, subprograms, and packages. If you mark an entity as Ghost, GNATprove ensures it can't affect the program's behavior. When the program is compiled with assertions enabled, ghost code is executed like normal code so it can execute the contracts using it. You can also instruct the compiler to not generate code for ghost entities.

Consider the procedure `Do_Something` below, which calls a complex function on its input, `X`, and wants to check that the initial and modified values of `X` are related in that complex way.

Listing 9: `show_ghost.ads`

```

1 package Show_Ghost is
2
3     type T is record
4         A, B, C, D, E : Boolean;
5     end record;
6
7     function Formula (X : T) return Boolean is
8         ((X.A and X.B) or (X.C and (X.D or X.E)));
9
10    function Is_Correct (X, Y : T) return Boolean is
11        (Formula (X) = Formula (Y));
12
13    procedure Do_Something (X : in out T);
14
15 end Show_Ghost;
```

Listing 10: `show_ghost.adb`

```

1 package body Show_Ghost is
2
3     procedure Do_Some_Complex_Stuff (X : in out T) is
4 begin
5     X := T'(X.B, X.A, X.C, X.E, X.D);
6 end Do_Some_Complex_Stuff;
7
8     procedure Do_Something (X : in out T) is
9         X_Init : constant T := X with Ghost;
10    begin
11        Do_Some_Complex_Stuff (X);
```

(continues on next page)

(continued from previous page)

```

12  pragma Assert (Is_Correct (X_Init, X));
13  -- It is OK to use X_Init inside an assertion.
14  end Do_Something;
15
16 end Show_Ghost;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_ghost.adb:12:22: info: assertion proved
```

Do_Something stores the initial value of X in a ghost constant, X_Init. We reference it in an assertion to check that the computation performed by the call to Do_Some_Complex_Stuff modified the value of X in the expected manner.

However, X_Init can't be used in normal code, for example to restore the initial value of X.

Listing 11: show_ghost.ads

```

1 package Show_Ghost is
2
3     type T is record
4         A, B, C, D, E : Boolean;
5     end record;
6
7     function Formula (X : T) return Boolean is
8         ((X.A and X.B) or (X.C and (X.D or X.E)));
9
10    function Is_Correct (X, Y : T) return Boolean is
11        (Formula (X) = Formula (Y));
12
13    procedure Do_Something (X : in out T);
14
15 end Show_Ghost;
```

Listing 12: show_ghost.adb

```

1 package body Show_Ghost is
2
3     procedure Do_Some_Complex_Stuff (X : in out T) is
4     begin
5         X := T'(X.B, X.A, X.C, X.E, X.D);
6     end Do_Some_Complex_Stuff;
7
8     procedure Do_Something (X : in out T) is
9         X_Init : constant T := X with Ghost;
10    begin
11        Do_Some_Complex_Stuff (X);
12        pragma Assert (Is_Correct (X_Init, X));
13
14        X := X_Init; -- ERROR
15
16    end Do_Something;
17
18 end Show_Ghost;
```

Listing 13: use_ghost.adb

```

1  with Show_Ghost; use Show_Ghost;
2
3  procedure Use_Ghost is
4      X : T := (True, True, False, False, True);
5  begin
6      Do_Something (X);
7  end Use_Ghost;

```

Build output

```

show_ghost.adb:14:12: error: ghost entity cannot appear in this context
gprbuild: *** compilation phase failed

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
show_ghost.adb:14:12: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

```

When compiling this example, the compiler flags the use of X_Init as illegal, but more complex cases of interference between ghost and normal code may sometimes only be detected when you run GNATprove.

29.2.2 Ghost Functions

Functions used only in specifications are a common occurrence when writing contracts for functional correctness. For example, expression functions used to simplify or factor out common patterns in contracts can usually be marked as ghost.

But ghost functions can do more than improve readability. In real-world programs, it's often the case that some information necessary for functional specification isn't accessible in the package's specification because of abstraction.

Making this information available to users of the packages is generally out of the question because that breaks the abstraction. Ghost functions come in handy in that case since they provide a way to give access to that information without making it available to normal client code.

Let's look at the following example.

Listing 14: stacks.ads

```

1  package Stacks is
2
3      pragma Unevaluated_Use_of_Old (Allow);
4
5      type Stack is private;
6
7      type Element is new Natural;
8      type Element_Array is array (Positive range <>) of Element;
9      Max : constant Natural := 100;
10
11     function Get_Model (S : Stack) return Element_Array with Ghost;
12     -- Returns an array as a model of a stack.
13
14     procedure Push (S : in out Stack; E : Element) with
15         Pre => Get_Model (S)'Length < Max,
16         Post => Get_Model (S) = Get_Model (S)'Old & E;

```

(continues on next page)

(continued from previous page)

```

17 private
18
19   subtype Length_Type is Natural range 0 .. Max;
20
21   type Stack is record
22     Top      : Length_Type := 0;
23     Content : Element_Array (1 .. Max) := (others => 0);
24   end record;
25
26
27 end Stacks;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
```

Here, the type Stack is private. To specify the expected behavior of the Push procedure, we need to go inside this abstraction and access the values of the elements stored in S. For this, we introduce a function Get_Model that returns an array as a representation of the stack. However, we don't want code that uses the Stack package to use Get_Model in normal code since this breaks our stack's abstraction.

Here's an example of trying to break that abstraction in the subprogram Peek below.

Listing 15: stacks.ads

```

1  package Stacks is
2
3   pragma Unevaluated_Use_of_Old (Allow);
4
5   type Stack is private;
6
7   type Element is new Natural;
8   type Element_Array is array (Positive range <>) of Element;
9   Max : constant Natural := 100;
10
11  function Get_Model (S : Stack) return Element_Array with Ghost;
12  -- Returns an array as a model of a stack.
13
14  procedure Push (S : in out Stack; E : Element) with
15    Pre => Get_Model (S)'Length < Max,
16    Post => Get_Model (S) = Get_Model (S)'Old & E;
17
18  function Peek (S : Stack; I : Positive) return Element is
19    (Get_Model (S) (I)); -- ERROR
20
21 private
22
23   subtype Length_Type is Natural range 0 .. Max;
24
25   type Stack is record
26     Top      : Length_Type := 0;
27     Content : Element_Array (1 .. Max) := (others => 0);
28   end record;
29
30 end Stacks;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
stacks.ads:19:07: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts
```

We see that marking the function as Ghost achieves this goal: it ensures that the subprogram Get_Model is never used in production code.

29.2.3 Global Ghost Variables

Though it happens less frequently, you may have specifications requiring you to store additional information in global variables that isn't needed in normal code. You should mark these global variables as ghost, allowing the compiler to remove them when assertions aren't enabled. You can use these variables for any purpose within the contracts that make up your specifications. A common scenario is writing specifications for subprograms that modify a complex or private global data structure: you can use these variables to provide a model for that structure that's updated by the ghost code as the program modifies the data structure itself.

You can also use ghost variables to store information about previous runs of subprograms to specify temporal properties. In the following example, we have two procedures, one that accesses a state A and the other that accesses a state B. We use the ghost variable Last_Accessed_Is_A to specify that B can't be accessed twice in a row without accessing A in between.

Listing 16: call_sequence.ads

```
1 package Call_Sequence is
2
3   type T is new Integer;
4
5   Last_Accessed_Is_A : Boolean := False with Ghost;
6
7   procedure Access_A with
8     Post => Last_Accessed_Is_A;
9
10  procedure Access_B with
11    Pre  => Last_Accessed_Is_A,
12    Post => not Last_Accessed_Is_A;
13    -- B can only be accessed after A
14
15 end Call_Sequence;
```

Listing 17: call_sequence.adb

```
1 package body Call_Sequence is
2
3   procedure Access_A is
4     begin
5       -- ...
6       Last_Accessed_Is_A := True;
7     end Access_A;
8
9   procedure Access_B is
10    begin
11      -- ...
12      Last_Accessed_Is_A := False;
13    end Access_B;
14
15 end Call_Sequence;
```

Listing 18: main.adb

```

1  with Call_Sequence; use Call_Sequence;
2
3  procedure Main is
4  begin
5      Access_A;
6      Access_B;
7      Access_B; -- ERROR
8  end Main;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
main.adb:7:04: medium: precondition might fail
gnatprove: unproved check messages considered as errors

```

Runtime output

```

raised ADA ASSERTIONS ASSERTION_ERROR : failed precondition from call_sequence.
  ↳ads:11

```

Let's look at another example. The specification of a subprogram's expected behavior is sometimes best expressed as a sequence of actions it must perform. You can use global ghost variables that store intermediate values of normal variables to write this sort of specification more easily.

For example, we specify the subprogram `Do_Two_Things` below in two steps, using the ghost variable `V_Interim` to store the intermediate value of `V` between those steps. We could also express this using an existential quantification on the variable `V_Interim`, but it would be impractical to iterate over all integers at runtime and this can't always be written in SPARK because quantification is restricted to `for ... loop` patterns.

Finally, supplying the value of the variable may help the prover verify the contracts.

Listing 19: action_sequence.ads

```

1  package Action_Sequence is
2
3      type T is new Integer;
4
5      V_Interim : T with Ghost;
6
7      function First_Thing_Done (X, Y : T) return Boolean with Ghost;
8      function Second_Thing_Done (X, Y : T) return Boolean with Ghost;
9
10     procedure Do_Two_Things (V : in out T) with
11         Post => First_Thing_Done (V'Old, V_Interim)
12             and then Second_Thing_Done (V_Interim, V);
13
14 end Action_Sequence;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

```

Note: For more details on ghost code, see the SPARK User's Guide⁴².

29.3 Guide Proof

Since properties of interest for functional correctness are more complex than those involved in proofs of program integrity, we expect GNATprove to initially be unable to verify them even though they're valid. You'll find the techniques we discussed in *Debugging Failed Proof Attempts* (page 314) to come in handy here. We now go beyond those techniques and focus on more ways of improving results in the cases where the property is valid but GNATprove can't prove it in a reasonable amount of time.

In those cases, you may want to try and guide GNATprove to either complete the proof or strip it down to a small number of easily-reviewable assumptions. For this purpose, you can add assertions to break complex proofs into smaller steps.

```
pragma Assert (Assertion_Checked_By_The_Tool);
-- info: assertion proved

pragma Assert (Assumption_Validated_By_Other_Means);
-- medium: assertion might fail

pragma Assume (Assumption_Validated_By_Other_Means);
-- The tool does not attempt to check this expression.
-- It is recorded as an assumption.
```

One such intermediate step you may find useful is to try to prove a theoretically-equivalent version of the desired property, but one where you've simplified things for the prover, such as by splitting up different cases or inlining the definitions of functions.

Some intermediate assertions may not be proved by GNATprove either because it's missing some information or because the amount of information available is confusing. You can verify these remaining assertions by other means such as testing (since they're executable) or by review. You can then choose to instruct GNATprove to ignore them, either by turning them into assumptions, as in our example, or by using a **pragma Annotate**. In both cases, the compiler generates code to check these assumptions at runtime when you enable assertions.

29.3.1 Local Ghost Variables

You can use ghost code to enhance what you can express inside intermediate assertions in the same way we did above to enhance our contracts in specifications. In particular, you'll commonly have local variables or constants whose only purpose is to be used in assertions. You'll mostly use these ghost variables to store previous values of variables or expressions you want to refer to in assertions. They're especially useful to refer to initial values of parameters and expressions since the '**Old**' attribute is only allowed in postconditions.

In the example below, we want to help GNATprove verify the postcondition of P. We do this by introducing a local ghost constant, X_Init, to represent this value and writing an assertion in both branches of an **if** statement that repeats the postcondition, but using X_Init.

⁴² https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/specification_features.html#ghost-code

Listing 20: show_local_ghost.ads

```

1 package Show_Local_Ghost is
2
3     type T is new Natural;
4
5     function F (X, Y : T) return Boolean is (X > Y) with Ghost;
6
7     function Condition (X : T) return Boolean is (X mod 2 = 0);
8
9     procedure P (X : in out T) with
10        Pre => X < 1_000_000,
11        Post => F (X, X'Old);
12
13 end Show_Local_Ghost;

```

Listing 21: show_local_ghost.adb

```

1 package body Show_Local_Ghost is
2
3     procedure P (X : in out T) is
4         X_Init : constant T := X with Ghost;
5     begin
6         if Condition (X) then
7             X := X + 1;
8             pragma Assert (F (X, X_Init));
9         else
10            X := X * 2;
11            pragma Assert (F (X, X_Init));
12        end if;
13    end P;
14
15 end Show_Local_Ghost;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_local_ghost.adb:7:17: info: overflow check proved
show_local_ghost.adb:8:25: info: assertion proved
show_local_ghost.adb:10:17: info: overflow check proved
show_local_ghost.adb:11:25: info: assertion proved
show_local_ghost.ads:7:52: info: division check proved
show_local_ghost.ads:11:14: info: postcondition proved

```

You can also use local ghost variables for more complex purposes such as building a data structure that serves as witness for a complex property of a subprogram. In our example, we want to prove that the Sort procedure doesn't create new elements, that is, that all the elements present in A after the sort were in A before the sort. This property isn't enough to ensure that a call to Sort produces a value for A that's a permutation of its value before the call (or that the values are indeed sorted). However, it's already complex for a prover to verify because it involves a nesting of quantifiers. To help GNATprove, you may find it useful to store, for each index I, an index J that has the expected property.

```

procedure Sort (A : in out Nat_Array) with
  Post => (for all I in A'Range =>
            (for some J in A'Range => A (I) = A'Old (J)))
is
  Permutation : Index_Array := (1 => 1, 2 => 2, ...) with Ghost;
begin
  ...

```

(continues on next page)

(continued from previous page)

```
end Sort;
```

29.3.2 Ghost Procedures

Ghost procedures can't affect the value of normal variables, so they're mostly used to perform operations on ghost variables or to group together a set of intermediate assertions.

Abstracting away the treatment of assertions and ghost variables inside a ghost procedure has several advantages. First, you're allowed to use these variables in any way you choose in code inside ghost procedures. This isn't the case outside ghost procedures, where the only ghost statements allowed are assignments to ghost variables and calls to ghost procedures.

As an example, the **for** loop contained in Increase_A couldn't appear by itself in normal code.

Listing 22: show_ghost_proc.ads

```
1 package Show_Ghost_Proc is
2
3     type Nat_Array is array (Integer range <>) of Natural;
4
5     A : Nat_Array (1 .. 100) with Ghost;
6
7     procedure Increase_A with
8         Ghost,
9         Pre => (for all I in A'Range => A (I) < Natural'Last);
10
11 end Show_Ghost_Proc;
```

Listing 23: show_ghost_proc.adb

```
1 package body Show_Ghost_Proc is
2
3     procedure Increase_A is
4     begin
5         for I in A'Range loop
6             A (I) := A (I) + 1;
7         end loop;
8     end Increase_A;
9
10 end Show_Ghost_Proc;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_ghost_proc.adb:6:25: info: overflow check proved
```

Using the abstraction also improves readability by hiding complex code that isn't part of the functional behavior of the subprogram. Finally, it can help GNATprove by abstracting away assertions that would otherwise make its job more complex.

In the example below, calling Prove_P with X as an operand only adds P (X) to the proof context instead of the larger set of assertions required to verify it. In addition, the proof of P need only be done once and may be made easier not having any unnecessary information present in its context while verifying it. Also, if GNATprove can't fully verify Prove_P, you can review the remaining assumptions more easily since they're in a smaller context.

```
procedure Prove_P (X : T) with Ghost,
  Global => null,
  Post   => P (X);
```

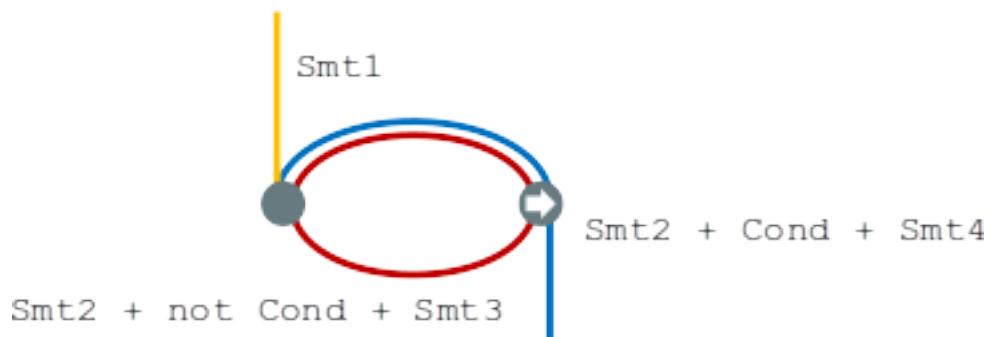
29.3.3 Handling of Loops

When the program involves a loop, you're almost always required to provide additional annotations to allow GNATprove to complete a proof because the verification techniques used by GNATprove don't handle cycles in a subprogram's control flow. Instead, loops are flattened by dividing them into several acyclic parts.

As an example, let's look at a simple loop with an exit condition.

```
Stmt1;
loop
  Stmt2;
  exit when Cond;
  Stmt3;
end loop;
Stmt4;
```

As shown below, the control flow is divided into three parts.



The first, shown in yellow, starts earlier in the subprogram and enters the loop statement. The loop itself is divided into two parts. Red represents a complete execution of the loop's body: an execution where the exit condition isn't satisfied. Blue represents the last execution of the loop, which includes some of the subprogram following it. For that path, the exit condition is assumed to hold. The red and blue parts are always executed after the yellow one.

GNATprove analyzes these parts independently since it doesn't have a way to track how variables may have been updated by an iteration of the loop. It forgets everything it knows about those variables from one part when entering another part. However, values of constants and variables that aren't modified in the loop are not an issue.

In other words, handling loops in that way makes GNATprove imprecise when verifying a subprogram involving a loop: it can't verify a property that relies on values of variables modified inside the loop. It won't forget any information it had on the value of constants or unmodified variables, but it nevertheless won't be able to deduce new information about them from the loop.

For example, consider the function *Find* which iterates over the array *A* and searches for an element where *E* is stored in *A*.

Listing 24: show_find.ads

```
1 package Show_Find is
2
```

(continues on next page)

(continued from previous page)

```

3  type Nat_Array is array (Positive range <>) of Natural;
4
5  function Find (A : Nat_Array; E : Natural) return Natural;
6
7 end Show_Find;

```

Listing 25: show_find.adb

```

1 package body Show_Find is
2
3   function Find (A : Nat_Array; E : Natural) return Natural is
4 begin
5   for I in A'Range loop
6     pragma Assert (for all J in A'First .. I - 1 => A (J) /= E);
7     -- assertion is not proved
8     if A (I) = E then
9       return I;
10    end if;
11    pragma Assert (A (I) /= E);
12    -- assertion is proved
13  end loop;
14  return 0;
15 end Find;
16
17 end Show_Find;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:6:51: info: overflow check proved
show_find.adb:6:58: medium: assertion might fail, cannot prove A (J) /= E_
  ↪[possible fix: subprogram at show_find.ads:5 should mention A and E in a_
  ↪precondition]
show_find.adb:6:61: info: index check proved
show_find.adb:9:20: info: range check proved
show_find.adb:11:25: info: assertion proved
gnatprove: unproved check messages considered as errors

```

At the end of each loop iteration, GNATprove knows that the value stored at index I in A must not be E. (If it were, the loop wouldn't have reached the end of the iteration.) This proves the second assertion. But it's unable to aggregate this information over multiple loop iterations to deduce that it's true for all the indexes smaller than I, so it can't prove the first assertion.

29.3.4 Loop Invariants

To overcome these limitations, you can provide additional information to GNATprove in the form of a *loop invariant*. In SPARK, a loop invariant is a Boolean expression which holds true at every iteration of the loop. Like other assertions, you can have it checked at runtime by compiling the program with assertions enabled.

The major difference between loop invariants and other assertions is the way it's treated for proofs. GNATprove performs the proof of a loop invariant in two steps: first, it checks that it holds for the first iteration of the loop and then it checks that it holds in an arbitrary iteration assuming it held in the previous iteration. This is called *proof by induction*⁴³.

⁴³ https://en.wikipedia.org/wiki/Mathematical_induction

As an example, let's add a loop invariant to the Find function stating that the first element of A is not E.

Listing 26: show_find.ads

```
1 package Show_Find is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Find (A : Nat_Array; E : Natural) return Natural;
6
7 end Show_Find;
```

Listing 27: show_find.adb

```
1 package body Show_Find is
2
3     function Find (A : Nat_Array; E : Natural) return Natural is
4 begin
5     for I in A'Range loop
6         pragma Loop_Invariant (A (A'First) /= E);
7         -- loop invariant not proved in first iteration
8         -- but preservation of loop invariant is proved
9         if A (I) = E then
10             return I;
11         end if;
12     end loop;
13     return 0;
14 end Find;
15
16 end Show_Find;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:6:33: info: loop invariant preservation proved
show_find.adb:6:33: medium: loop invariant might fail in first iteration [possible_
fix: subprogram at show_find.ads:5 should mention A and E in a precondition]
show_find.adb:6:37: info: index check proved
show_find.adb:10:20: info: range check proved
gnatprove: unproved check messages considered as errors
```

To verify this invariant, GNATprove generates two checks. The first checks that the assertion holds in the first iteration of the loop. This isn't verified by GNATprove. And indeed there's no reason to expect the first element of A to always be different from E in this iteration. However, the second check is proved: it's easy to deduce that if the first element of A was not E in a given iteration it's still not E in the next. However, if we move the invariant to the end of the loop, then it is successfully verified by GNATprove.

Not only do loop invariants allow you to verify complex properties of loops, but GNATprove also uses them to verify other properties, such as the absence of runtime errors over both the loop's body and the statements following the loop. More precisely, when verifying a runtime check or other assertion there, GNATprove assumes that the last occurrence of the loop invariant preceding the check or assertion is true.

Let's look at a version of Find where we use a loop invariant instead of an assertion to state that none of the array elements seen so far are equal to E.

Listing 28: show_find.ads

```

1 package Show_Find is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function Find (A : Nat_Array; E : Natural) return Natural;
6
7 end Show_Find;

```

Listing 29: show_find.adb

```

1 package body Show_Find is
2
3   function Find (A : Nat_Array; E : Natural) return Natural is
4     begin
5       for I in A'Range loop
6         pragma Loop_Invariant
7           (for all J in A'First .. I - 1 => A (J) /= E);
8         if A (I) = E then
9           return I;
10        end if;
11      end loop;
12      pragma Assert (for all I in A'Range => A (I) /= E);
13      return 0;
14    end Find;
15
16 end Show_Find;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_find.adb:7:13: info: loop invariant initialization proved
show_find.adb:7:13: info: loop invariant preservation proved
show_find.adb:7:39: info: overflow check proved
show_find.adb:7:49: info: index check proved
show_find.adb:9:20: info: range check proved
show_find.adb:12:22: info: assertion proved
show_find.adb:12:49: info: index check proved

```

This version is fully verified by GNATprove! This time, it proves that the loop invariant holds in every iteration of the loop (separately proving this property for the first iteration and then for the following iterations). It also proves that none of the elements of A are equal to E after the loop exits by assuming that the loop invariant holds in the last iteration of the loop.

Note: For more details on loop invariants, see the SPARK User's Guide⁴⁴.

Finding a good loop invariant can turn out to be quite a challenge. To make this task easier, let's review the four good properties of a good loop invariant:

⁴⁴ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/assertion_pragmas.html#loop-invariants

Prop- erty	Description
INIT	It should be provable in the first iteration of the loop.
INSIDE	It should allow proving the absence of run-time errors and local assertions inside the loop.
AFTER	It should allow proving absence of run-time errors, local assertions, and the subprogram postcondition after the loop.
PRE- SERVE	It should be provable after the first iteration of the loop.

Let's look at each of these in turn. First, the loop invariant should be provable in the first iteration of the loop (INIT). If your invariant fails to achieve this property, you can debug the loop invariant's initialization like any failing proof attempt using strategies for [Debugging Failed Proof Attempts](#) (page 314).

Second, the loop invariant should be precise enough to allow GNATprove to prove absence of runtime errors in both statements from the loop's body (INSIDE) and those following the loop (AFTER). To do this, you should remember that all information concerning a variable modified in the loop that's not included in the invariant is forgotten by GNATprove. In particular, you should take care to include in your invariant what's usually called the loop's *frame condition*, which lists properties of variables that are true throughout the execution of the loop even though those variables are modified by the loop.

Finally, the loop invariant should be precise enough to prove that it's preserved through successive iterations of the loop (PRESERVE). This is generally the trickiest part. To understand why GNATprove hasn't been able to verify the preservation of a loop invariant you provided, you may find it useful to repeat it as local assertions throughout the loop's body to determine at which point it can no longer be proved.

As an example, let's look at a loop that iterates through an array A and applies a function F to each of its elements.

Listing 30: show_map.ads

```

1 package Show_Map is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function F (V : Natural) return Natural is
6         (if V /= Natural'Last then V + 1 else V);
7
8     procedure Map (A : in out Nat_Array);
9
10 end Show_Map;
```

Listing 31: show_map.adb

```

1 package body Show_Map is
2
3     procedure Map (A : in out Nat_Array) is
4         A_I : constant Nat_Array := A with Ghost;
5     begin
6         for K in A'Range loop
7             A (K) := F (A (K));
8             pragma Loop_Invariant
9                 (for all J in A'First .. K => A (J) = F (A'Loop_Entry (J)));
10            end loop;
11            pragma Assert (for all K in A'Range => A (K) = F (A_I (K)));
12        end Map;
```

(continues on next page)

(continued from previous page)

```
14 end Show_Map;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_map.adb:9:13: info: loop invariant initialization proved
show_map.adb:9:13: info: loop invariant preservation proved
show_map.adb:9:45: info: index check proved
show_map.adb:9:67: info: index check proved
show_map.adb:11:22: info: assertion proved
show_map.adb:11:49: info: index check proved
show_map.adb:11:62: info: index check proved
show_map.ads:6:35: info: overflow check proved
```

After the loop, each element of A should be the result of applying F to its previous value. We want to prove this. To specify this property, we copy the value of A before the loop into a ghost variable, A_I. Our loop invariant states that the element at each index less than K has been modified in the expected way. We use the Loop_Entry attribute to refer to the value of A on entry of the loop instead of using A_I.

Does our loop invariant have the four properties of a good loop-invariant? When launching GNATprove, we see that INIT is fulfilled: the invariant's initialization is proved. So are INSIDE and AFTER: no potential runtime errors are reported and the assertion following the loop is successfully verified.

The situation is slightly more complex for the PRESERVE property. GNATprove manages to prove that the invariant holds after the first iteration thanks to the automatic generation of frame conditions. It was able to do this because it completes the provided loop invariant with the following frame condition stating what part of the array hasn't been modified so far:

```
pragma Loop_Invariant
  (for all J in K .. A'Last => A (J) = (if J > K then A'Loop_Entry (J)));
```

GNATprove then uses both our and the internally-generated loop invariants to prove PRESERVE. However, in more complex cases, the heuristics used by GNATprove to generate the frame condition may not be sufficient and you'll have to provide one as a loop invariant. For example, consider a version of Map where the result of applying F to an element at index K is stored at index K-1:

Listing 32: show_map.ads

```
1 package Show_Map is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   function F (V : Natural) return Natural is
6     (if V /= Natural'Last then V + 1 else V);
7
8   procedure Map (A : in out Nat_Array);
9
10 end Show_Map;
```

Listing 33: show_map.adb

```
1 package body Show_Map is
2
3   procedure Map (A : in out Nat_Array) is
4     A_I : constant Nat_Array := A with Ghost;
```

(continues on next page)

(continued from previous page)

```

5   begin
6     for K in A'Range loop
7       if K /= A'First then
8         A (K - 1) := F (A (K));
9       end if;
10      pragma Loop_Invariant
11        (for all J in A'First .. K =>
12          (if J /= A'First then A (J - 1) = F (A'Loop_Entry (J))));
13        -- pragma Loop_Invariant
14        -- (for all J in K .. A'Last => A (J) = A'Loop_Entry (J));
15      end loop;
16      pragma Assert (for all K in A'Range =>
17                      (if K /= A'First then A (K - 1) = F (A_I (K))));
18    end Map;
19
20  end Show_Map;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_map.adb:8:18: info: overflow check proved
show_map.adb:8:18: info: index check proved
show_map.adb:11:13: info: loop invariant initialization proved
show_map.adb:12:36: medium: loop invariant might not be preserved by an arbitrary
iteration, cannot prove A (J - 1) = F (A'Loop_Entry (J))
show_map.adb:12:41: info: overflow check proved
show_map.adb:12:41: info: index check proved
show_map.adb:12:65: info: index check proved
show_map.adb:16:22: info: assertion proved
show_map.adb:17:50: info: overflow check proved
show_map.adb:17:50: info: index check proved
show_map.adb:17:65: info: index check proved
show_map.ads:6:35: info: overflow check proved
gnatprove: unproved check messages considered as errors

```

You need to uncomment the second loop invariant containing the frame condition in order to prove the assertion after the loop.

Note: For more details on how to write a loop invariant, see the SPARK User's Guide⁴⁵.

⁴⁵ https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_write_loop_invariants.html

29.4 Code Examples / Pitfalls

This section contains some code examples and pitfalls.

29.4.1 Example #1

We implement a ring buffer inside an array Content, where the contents of a ring buffer of length Length are obtained by starting at index First and possibly wrapping around the end of the buffer. We use a ghost function Get_Model to return the contents of the ring buffer for use in contracts.

Listing 34: ring_buffer.ads

```

1 package Ring_Buffer is
2
3     Max_Size : constant := 100;
4
5     type Nat_Array is array (Positive range <>) of Natural;
6
7     function Get_Model return Nat_Array with Ghost;
8
9     procedure Push_Last (E : Natural) with
10        Pre => Get_Model'Length < Max_Size,
11        Post => Get_Model'Length = Get_Model'Old'Length + 1;
12
13 end Ring_Buffer;
```

Listing 35: ring_buffer.adb

```

1 package body Ring_Buffer is
2
3     subtype Length_Range is Natural range 0 .. Max_Size;
4     subtype Index_Range is Natural range 1 .. Max_Size;
5
6     Content : Nat_Array (1 .. Max_Size) := (others => 0);
7     First   : Index_Range           := 1;
8     Length  : Length_Range         := 0;
9
10    function Get_Model return Nat_Array with
11        Refined_Post => Get_Model'Result'Length = Length
12    is
13        Size   : constant Length_Range := Length;
14        Result : Nat_Array (1 .. Size) := (others => 0);
15    begin
16        if First + Length - 1 <= Max_Size then
17            Result := Content (First .. First + Length - 1);
18        else
19            declare
20                Len : constant Length_Range := Max_Size - First + 1;
21            begin
22                Result (1 .. Len) := Content (First .. Max_Size);
23                Result (Len + 1 .. Length) := Content (1 .. Length - Len);
24            end;
25        end if;
26        return Result;
27    end Get_Model;
```

(continues on next page)

(continued from previous page)

```

28
29  procedure Push_Last (E : Natural) is
30  begin
31      if First + Length <= Max_Size then
32          Content (First + Length) := E;
33      else
34          Content (Length - Max_Size + First) := E;
35      end if;
36      Length := Length + 1;
37  end Push_Last;
38
39 end Ring_Buffer;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
ring_buffer.adb:7:04: warning: "First" is not modified, could be declared constant_gnatwk
ring_buffer.adb:11:22: info: refined post proved
ring_buffer.adb:11:38: info: range check proved
ring_buffer.adb:14:07: info: range check proved
ring_buffer.adb:14:41: info: length check proved
ring_buffer.adb:17:17: info: length check proved
ring_buffer.adb:17:20: info: range check proved
ring_buffer.adb:17:20: info: length check proved
ring_buffer.adb:20:61: info: range check proved
ring_buffer.adb:22:13: info: range check proved
ring_buffer.adb:22:31: info: length check proved
ring_buffer.adb:22:34: info: range check proved
ring_buffer.adb:22:34: info: length check proved
ring_buffer.adb:23:13: info: range check proved
ring_buffer.adb:23:40: info: length check proved
ring_buffer.adb:23:43: info: range check proved
ring_buffer.adb:23:43: info: length check proved
ring_buffer.adb:32:25: info: index check proved
ring_buffer.adb:34:37: info: index check proved
ring_buffer.adb:36:24: info: range check proved
ring_buffer.ads:11:14: info: postcondition proved
```

This is correct: Get_Model is used only in contracts. Calls to Get_Model make copies of the buffer's contents, which isn't efficient, but is fine because Get_Model is only used for verification, not in production code. We enforce this by making it a ghost function. We'll produce the final production code with appropriate compiler switches (i.e., not using -gnata) that ensure assertions are ignored.

29.4.2 Example #2

Instead of using a ghost function, Get_Model, to retrieve the contents of the ring buffer, we're now using a global ghost variable, Model.

Listing 36: ring_buffer.ads

```

1 package Ring_Buffer is
2
3     Max_Size : constant := 100;
4     subtype Length_Range is Natural range 0 .. Max_Size;
5     subtype Index_Range is Natural range 1 .. Max_Size;
6
```

(continues on next page)

(continued from previous page)

```

7  type Nat_Array is array (Positive range <>) of Natural;
8
9  type Model_Type (Length : Length_Range := 0) is record
10   Content : Nat_Array (1 .. Length);
11 end record;
12 with Ghost;
13
14 Model : Model_Type with Ghost;
15
16 function Valid_Model return Boolean;
17
18 procedure Push_Last (E : Natural) with
19   Pre => Valid_Model
20   and then Model.Length < Max_Size,
21   Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;

```

Listing 37: ring_buffer.adb

```

1 package body Ring_Buffer is
2
3   Content : Nat_Array (1 .. Max_Size) := (others => 0);
4   First   : Index_Range           := 1;
5   Length  : Length_Range         := 0;
6
7   function Valid_Model return Boolean is
8     (Model.Content'Length = Length);
9
10  procedure Push_Last (E : Natural) is
11 begin
12   if First + Length <= Max_Size then
13     Content (First + Length) := E;
14   else
15     Content (Length - Max_Size + First) := E;
16   end if;
17   Length := Length + 1;
18 end Push_Last;
19
20 end Ring_Buffer;

```

Build output

```

ring_buffer.adb:8:08: error: ghost entity cannot appear in this context
gprbuild: *** compilation phase failed

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
ring_buffer.adb:8:08: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts

```

This example isn't correct. `Model`, which is a ghost variable, must not influence the return value of the normal function `Valid_Model`. Since `Valid_Model` is only used in specifications, we should have marked it as `Ghost`. Another problem is that `Model` needs to be updated inside `Push_Last` to reflect the changes to the ring buffer.

29.4.3 Example #3

Let's mark `Valid_Model` as Ghost and update `Model` inside `Push_Last`.

Listing 38: `ring_buffer.ads`

```

1 package Ring_Buffer is
2
3     Max_Size : constant := 100;
4     subtype Length_Range is Natural range 0 .. Max_Size;
5     subtype Index_Range  is Natural range 1 .. Max_Size;
6
7     type Nat_Array is array (Positive range <>) of Natural;
8
9     type Model_Type (Length : Length_Range := 0) is record
10        Content : Nat_Array (1 .. Length);
11    end record
12    with Ghost;
13
14    Model : Model_Type with Ghost;
15
16    function Valid_Model return Boolean with Ghost;
17
18    procedure Push_Last (E : Natural) with
19        Pre => Valid_Model
20        and then Model.Length < Max_Size,
21        Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;
```

Listing 39: `ring_buffer.adb`

```

1 package body Ring_Buffer is
2
3     Content : Nat_Array (1 .. Max_Size) := (others => 0);
4     First   : Index_Range              := 1;
5     Length  : Length_Range            := 0;
6
7     function Valid_Model return Boolean is
8         (Model.Content'Length = Length);
9
10    procedure Push_Last (E : Natural) is
11    begin
12        if First + Length <= Max_Size then
13            Content (First + Length) := E;
14        else
15            Content (Length - Max_Size + First) := E;
16        end if;
17        Length := Length + 1;
18        Model := (Length => Model.Length + 1,
19                   Content => Model.Content & E);
20    end Push_Last;
21
22 end Ring_Buffer;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
ring_buffer.adb:3:04: warning: variable "Content" is assigned but never read [-
    -gnatwm]
ring_buffer.adb:4:04: warning: "First" is not modified, could be declared constant
(continues on next page)
```

(continued from previous page)

```

→ [-gnatwk]
ring_buffer.adb:8:21: info: range check proved
ring_buffer.adb:13:25: info: index check proved
ring_buffer.adb:15:37: info: index check proved
ring_buffer.adb:17:24: info: range check proved
ring_buffer.adb:18:13: info: discriminant check proved
ring_buffer.adb:18:41: info: range check proved
ring_buffer.adb:19:42: info: range check proved
ring_buffer.adb:19:42: info: length check proved
ring_buffer.ads:10:07: info: range check proved
ring_buffer.ads:21:14: info: postcondition proved

```

This example is correct. The ghost variable `Model` can be referenced both from the body of the ghost function `Valid_Model` and the non-ghost procedure `Push_Last` as long as it's only used in ghost statements.

29.4.4 Example #4

We're now modifying `Push_Last` to share the computation of the new length between the operational and ghost code.

Listing 40: `ring_buffer.ads`

```

1 package Ring_Buffer is
2
3   Max_Size : constant := 100;
4   subtype Length_Range is Natural range 0 .. Max_Size;
5   subtype Index_Range  is Natural range 1 .. Max_Size;
6
7   type Nat_Array is array (Positive range <>) of Natural;
8
9   type Model_Type (Length : Length_Range := 0) is record
10     Content : Nat_Array (1 .. Length);
11   end record;
12   with Ghost;
13
14   Model : Model_Type with Ghost;
15
16   function Valid_Model return Boolean with Ghost;
17
18   procedure Push_Last (E : Natural) with
19     Pre => Valid_Model
20     and then Model.Length < Max_Size,
21     Post => Model.Length = Model.Length'Old + 1;
22
23 end Ring_Buffer;

```

Listing 41: `ring_buffer.adb`

```

1 package body Ring_Buffer is
2
3   Content : Nat_Array (1 .. Max_Size) := (others => 0);
4   First   : Index_Range              := 1;
5   Length  : Length_Range            := 0;
6
7   function Valid_Model return Boolean is
8     (Model.Content'Length = Length);
9
10  procedure Push_Last (E : Natural) is

```

(continues on next page)

(continued from previous page)

```

11  New_Length : constant Length_Range := Model.Length + 1;
12 begin
13   if First + Length <= Max_Size then
14     Content (First + Length) := E;
15   else
16     Content (Length - Max_Size + First) := E;
17   end if;
18   Length := New_Length;
19   Model := (Length => New_Length,
20             Content => Model.Content & E);
21 end Push_Last;
22
23 end Ring_Buffer;
```

Build output

```
ring_buffer.adb:11:45: error: ghost entity cannot appear in this context
gprbuild: *** compilation phase failed
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
ring_buffer.adb:11:45: error: ghost entity cannot appear in this context
gnatprove: error during generation of Global contracts
```

This example isn't correct. We didn't mark local constant New_Length as Ghost, so it can't be computed from the value of ghost variable Model. If we made New_Length a ghost constant, the compiler would report the problem on the assignment from New_Length to Length. The correct solution here is to compute New_Length from the value of the non-ghost variable Length.

29.4.5 Example #5

Let's move the code updating Model inside a local ghost procedure, Update_Model, but still using a local variable, New_Length, to compute the length.

Listing 42: ring_buffer.ads

```

1  package Ring_Buffer is
2
3    Max_Size : constant := 100;
4    subtype Length_Range is Natural range 0 .. Max_Size;
5    subtype Index_Range is Natural range 1 .. Max_Size;
6
7    type Nat_Array is array (Positive range <>) of Natural;
8
9    type Model_Type (Length : Length_Range := 0) is record
10      Content : Nat_Array (1 .. Length);
11    end record;
12    with Ghost;
13
14    Model : Model_Type with Ghost;
15
16    function Valid_Model return Boolean with Ghost;
17
18    procedure Push_Last (E : Natural) with
19      Pre => Valid_Model
20      and then Model.Length < Max_Size,
21      Post => Model.Length = Model.Length'Old + 1;
```

(continues on next page)

(continued from previous page)

```
22
23 end Ring_Buffer;
```

Listing 43: ring_buffer.adb

```
1 package body Ring_Buffer is
2
3   Content : Nat_Array (1 .. Max_Size) := (others => 0);
4   First   : Index_Range           := 1;
5   Length  : Length_Range         := 0;
6
7   function Valid_Model return Boolean is
8     (Model.Content'Length = Length);
9
10  procedure Push_Last (E : Natural) is
11
12    procedure Update_Model with Ghost is
13      New_Length : constant Length_Range := Model.Length + 1;
14    begin
15      Model := (Length => New_Length,
16                 Content => Model.Content & E);
17    end Update_Model;
18
19  begin
20    if First + Length <= Max_Size then
21      Content (First + Length) := E;
22    else
23      Content (Length - Max_Size + First) := E;
24    end if;
25    Length := Length + 1;
26    Update_Model;
27  end Push_Last;
28
29 end Ring_Buffer;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
ring_buffer.adb:3:04: warning: variable "Content" is assigned but never read [-
  ↵gnatwm]
ring_buffer.adb:4:04: warning: "First" is not modified, could be declared constant
  ↵[-gnatwk]
ring_buffer.adb:8:21: info: range check proved
ring_buffer.adb:13:61: info: range check proved, in call inlined at ring_buffer.
  ↵adb:26
ring_buffer.adb:15:16: info: discriminant check proved, in call inlined at ring_
  ↵buffer.adb:26
ring_buffer.adb:16:45: info: range check proved, in call inlined at ring_buffer.
  ↵adb:26
ring_buffer.adb:16:45: info: length check proved, in call inlined at ring_buffer.
  ↵adb:26
ring_buffer.adb:21:25: info: index check proved
ring_buffer.adb:23:37: info: index check proved
ring_buffer.adb:25:24: info: range check proved
ring_buffer.ads:10:07: info: range check proved
ring_buffer.ads:21:14: info: postcondition proved
```

Everything's fine here. Model is only accessed inside Update_Model, itself a ghost procedure, so it's fine to declare local variable New_Length without the Ghost aspect: everything inside a ghost procedure body is ghost. Moreover, we don't need to add any contract to

Update_Model: it's inlined by GNATprove because it's a local procedure without a contract.

29.4.6 Example #6

The function Max_Array takes two arrays of the same length (but not necessarily with the same bounds) as arguments and returns an array with each entry being the maximum values of both arguments at that index.

Listing 44: array_util.ads

```

1 package Array_Util is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Max_Array (A, B : Nat_Array) return Nat_Array with
6         Pre => A'Length = B'Length;
7
8 end Array_Util;
```

Listing 45: array_util.adb

```

1 package body Array_Util is
2
3     function Max_Array (A, B : Nat_Array) return Nat_Array is
4         R : Nat_Array (A'Range);
5         J : Integer := B'First;
6     begin
7         for I in A'Range loop
8             if A (I) > B (J) then
9                 R (I) := A (I);
10            else
11                R (I) := B (J);
12            end if;
13            J := J + 1;
14        end loop;
15        return R;
16    end Max_Array;
17
18 end Array_Util;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
array_util.adb:8:24: medium: array index check might fail [reason for check: value ↴
   must be a valid index into the array] [possible fix: loop at line 7 should ↴
   mention J in a loop invariant]
array_util.adb:13:17: medium: overflow check might fail, cannot prove upper bound ↴
   for J + 1 [reason for check: result of addition must fit in a 32-bits machine ↴
   integer] [possible fix: loop at line 7 should mention J in a loop invariant]
gnatprove: unproved check messages considered as errors
```

This program is correct, but GNATprove can't prove that J is always in the index range of B (the unproved index check) or even that it's always within the bounds of its type (the unproved overflow check). Indeed, when checking the body of the loop, GNATprove forgets everything about the current value of J because it's been modified by previous loop iterations. To get more precise results, we need to provide a loop invariant.

29.4.7 Example #7

Let's add a loop invariant that states that J stays in the index range of B and let's protect the increment to J by checking that it's not already the maximal integer value.

Listing 46: array_util.ads

```

1 package Array_Util is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Max_Array (A, B : Nat_Array) return Nat_Array with
6         Pre => A'Length = B'Length;
7
8 end Array_Util;
```

Listing 47: array_util.adb

```

1 package body Array_Util is
2
3     function Max_Array (A, B : Nat_Array) return Nat_Array is
4         R : Nat_Array (A'Range);
5         J : Integer := B'First;
6
7     begin
8         for I in A'Range loop
9             pragma Loop_Invariant (J in B'Range);
10            if A (I) > B (J) then
11                R (I) := A (I);
12            else
13                R (I) := B (J);
14            end if;
15            if J < Integer'Last then
16                J := J + 1;
17            end if;
18        end loop;
19        return R;
20    end Max_Array;
21
22 end Array_Util;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
array_util.adb:8:33: medium: loop invariant might not be preserved by an arbitrary
iteration
gnatprove: unproved check messages considered as errors
```

The loop invariant now allows verifying that no runtime error can occur in the loop's body (property INSIDE seen in section *Loop Invariants* (page 371)). Unfortunately, GNATprove fails to verify that the invariant stays valid after the first iteration of the loop (property PRESERVE). Indeed, knowing that J is in B'Range in a given iteration isn't enough to prove it'll remain so in the next iteration. We need a more precise invariant, linking J to the value of the loop index I, like $J = I - A'\text{First} + B'\text{First}$.

29.4.8 Example #8

We now consider a version of Max_Array which takes arguments that have the same bounds. We want to prove that Max_Array returns an array of the maximum values of both its arguments at each index.

Listing 48: array_util.ads

```

1 package Array_Util is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     function Max_Array (A, B : Nat_Array) return Nat_Array with
6         Pre => A'First = B'First and A'Last = B'Last,
7         Post => (for all K in A'Range =>
8                     Max_Array'Result (K) = Natural'Max (A (K), B (K)));
9
10    end Array_Util;

```

Listing 49: array_util.adb

```

1 package body Array_Util is
2
3     function Max_Array (A, B : Nat_Array) return Nat_Array is
4         R : Nat_Array (A'Range) := (others => 0);
5     begin
6         for I in A'Range loop
7             pragma Loop_Invariant (for all K in A'First .. I =>
8                                     R (K) = Natural'Max (A (K), B (K)));
9             if A (I) > B (I) then
10                 R (I) := A (I);
11             else
12                 R (I) := B (I);
13             end if;
14         end loop;
15         return R;
16     end Max_Array;
17
18    end Array_Util;

```

Listing 50: main.adb

```

1 with Array_Util; use Array_Util;
2
3 procedure Main is
4     A : Nat_Array := (1, 1, 2);
5     B : Nat_Array := (2, 1, 0);
6     R : Nat_Array (1 .. 3);
7 begin
8     R := Max_Array (A, B);
9 end Main;

```

Build output

```

main.adb:4:04: warning: "A" is not modified, could be declared constant [-gnatwk]
main.adb:5:04: warning: "B" is not modified, could be declared constant [-gnatwk]
main.adb:6:04: warning: variable "R" is assigned but never read [-gnatwm]
main.adb:8:04: warning: possibly useless assignment to "R", value might not be
→ referenced [-gnatwm]

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
main.adb:4:04: warning: "A" is not modified, could be declared constant [-gnatwk]
main.adb:5:04: warning: "B" is not modified, could be declared constant [-gnatwk]
main.adb:6:04: warning: variable "R" is assigned but never read [-gnatwm]
main.adb:8:04: warning: possibly useless assignment to "R", value might not be_
    ↴referenced [-gnatwm]
main.adb:8:09: medium: length check might fail [reason for check: array must be of_
    ↴the appropriate length]
array_util.adb:8:35: medium: loop invariant might not be preserved by an arbitrary_
    ↴iteration, cannot prove R (K) = Natural'max
array_util.adb:8:35: medium: loop invariant might fail in first iteration, cannot_
    ↴prove R (K) = Natural'max
gnatprove: unproved check messages considered as errors

```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : Loop_Invariant failed at array_util.adb:7
```

Here, GNATprove doesn't manage to prove the loop invariant even for the first loop iteration (property INIT seen in section *Loop Invariants* (page 371)). In fact, the loop invariant is incorrect, as you can see by executing the function Max_Array with assertions enabled: at each loop iteration, R contains the maximum of A and B only until $I - 1$ because the I 'th index wasn't yet handled.

29.4.9 Example #9

We now consider a procedural version of Max_Array which updates its first argument instead of returning a new array. We want to prove that Max_Array sets the maximum values of both its arguments into each index in its first argument.

Listing 51: array_util.ads

```

1 package Array_Util is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
6         Pre => A'First = B'First and A'Last = B'Last,
7         Post => (for all K in A'Range =>
8                 A (K) = Natural'Max (A'Old (K), B (K)));
9
10 end Array_Util;

```

Listing 52: array_util.adb

```

1 package body Array_Util is
2
3     procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
4 begin
5     for I in A'Range loop
6         pragma Loop_Invariant
7             (for all K in A'First .. I - 1 =>
8                 A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
9         pragma Loop_Invariant
10            (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
11         if A (I) <= B (I) then
12             A (I) := B (I);

```

(continues on next page)

(continued from previous page)

```

13      end if;
14   end loop;
15 end Max_Array;
16
17 end Array_Util;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
array_util.adb:7:13: info: loop invariant preservation proved
array_util.adb:7:13: info: loop invariant initialization proved
array_util.adb:7:39: info: overflow check proved
array_util.adb:8:18: info: index check proved
array_util.adb:8:50: info: index check proved
array_util.adb:8:57: info: index check proved
array_util.adb:10:13: info: loop invariant initialization proved
array_util.adb:10:13: info: loop invariant preservation proved
array_util.adb:10:44: info: index check proved
array_util.adb:10:63: info: index check proved
array_util.adb:11:25: info: index check proved
array_util.adb:12:25: info: index check proved
array_util.ads:7:14: info: postcondition proved
array_util.ads:8:20: info: index check proved
array_util.ads:8:45: info: index check proved
array_util.ads:8:52: info: index check proved
```

Everything is proved. The first loop invariant states that the values of A before the loop index contains the maximum values of the arguments of Max_Array (referring to the input value of A with A'Loop_Entry). The second loop invariant states that the values of A beyond and including the loop index are the same as they were on entry. This is the frame condition of the loop.

29.4.10 Example #10

Let's remove the frame condition from the previous example.

Listing 53: array_util.ads

```

1 package Array_Util is
2
3   type Nat_Array is array (Positive range <>) of Natural;
4
5   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
6     Pre => A'First = B'First and A'Last = B>Last,
7     Post => (for all K in A'Range =>
8                 A (K) = Natural'Max (A'Old (K), B (K)));
9
10 end Array_Util;
```

Listing 54: array_util.adb

```

1 package body Array_Util is
2
3   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
4 begin
5   for I in A'Range loop
6     pragma Loop_Invariant
```

(continues on next page)

(continued from previous page)

```

7   (for all K in A'First .. I - 1 =>
8     A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
9   if A (I) <= B (I) then
10    A (I) := B (I);
11   end if;
12 end loop;
13 end Max_Array;
14
15 end Array_Util;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
array_util.adb:7:13: info: loop invariant initialization proved
array_util.adb:7:13: info: loop invariant preservation proved
array_util.adb:7:39: info: overflow check proved
array_util.adb:8:18: info: index check proved
array_util.adb:8:50: info: index check proved
array_util.adb:8:57: info: index check proved
array_util.adb:9:25: info: index check proved
array_util.adb:10:25: info: index check proved
array_util.ads:7:14: info: postcondition proved
array_util.ads:8:20: info: index check proved
array_util.ads:8:45: info: index check proved
array_util.ads:8:52: info: index check proved

```

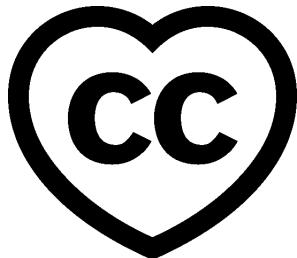
Everything is still proved. GNATprove internally generates the frame condition for the loop, so it's sufficient here to state that A before the loop index contains the maximum values of the arguments of Max_Array.

Part III

Introduction to Embedded Systems Programming

Copyright © 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page⁴⁶](#)



This course will teach you the basics of the Embedded Systems Programming using Ada.

This document was written by Patrick Rogers, with review by Stephen Baird, Tucker Taft, Filip Gajowniczek, and Gustavo A. Hoffmann.

⁴⁶ <http://creativecommons.org/licenses/by-sa/4.0>

INTRODUCTION

This is a course about embedded systems programming. Embedded systems are everywhere today, including — just to name a few — the thermostats that control a building's temperature, the power-steering controller in modern automobiles, and the control systems in charge of jet engines.

Clearly, much can depend on these systems operating correctly. It might be only a matter of comfort if the thermostat fails. But imagine what might happen if one of the critical control systems in your car failed when you're out on the freeway. When a jet engine controller is designed to have absolute control, it is known as a Full Authority Digital Engine Controller, or FADEC for short. If a FADEC fails, the result can make international news.

Using Ada can help you get it right, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal is to get you started in using it well.

The course is based on the assumption that you know some of the Ada language already, preferably even some of the more advanced concepts. You don't need to know how to use Ada constructs for embedded systems, of course, but you do need to know at least the language basics. If you need that introduction, see the course *Introduction to Ada* (page 5).

We also assume that you already have some programming experience so we won't cover CS-101.

Ideally, you also have some experience with low-level programming, because we will focus on "how to do it in Ada." If you do, feel free to gloss over the introductory material. If not, don't worry. We will cover enough for the course to be of value in any case.

30.1 So, what will we actually cover?

We will introduce you to using Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data, e.g., data from sensors that are occasionally faulty.

We will discuss development using more than Ada alone, nowadays a quite common approach. Specifically, how to interface with code and data written in other languages, and how (and why) to work with assembly language.

Embedded systems interact with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. Frequently these devices are mapped into the target memory address space. We will cover how to define and interact with these memory-mapped devices.

Finally, we will show how to handle interrupts in Ada, using portable constructs.

30.2 Definitions

Before we go any further, what do we mean by "embedded system" anyway? It's time to be specific. We're talking about a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system: the enclosing thermostat controlling the temperature, the power steering controller in the enclosing automobile, and the FADEC embedded in the enclosing aircraft. So these are not stand-alone computers for general purpose application execution.

As such, embedded systems typically have reduced resources available, especially power, which means reduced processor speed and reduced memory on-board. For an example at the small end of the spectrum, consider the computer embedded in a wearable device: it must run for a long time on a very little battery, with comparatively little memory available. But that's often true of bigger systems too, such as systems on aircraft where power (and heat) are directly limiting factors.

As a result, developing embedded systems software can be more difficult than general application development, not to mention that this software is potentially safety-critical.

Ada is known for use in very large, very long-lived projects (e.g., deployed for decades), but it can also be used for very small systems with tight resource constraints. We'll show you how.

We used the term "computer" above. You already know what that means, but you may be thinking of your laptop or something like that, where the processor, memory, and devices are all distinct, separate components. That can be the case for embedded systems too, albeit in a different form-factor such as rack-mounted boards. However, be sure to expand your definition to include the notion of a system-on-chip (SoC), in which the processor, memory, and various useful devices are all on a single chip. Embedded systems don't necessarily involve SoC computers but they frequently do. The techniques and information in this course work on any of these kinds of computer.

30.3 Down To The Bare Metal

Ada has always had facilities designed specifically for embedded systems. The language includes constructs for directly manipulating hardware, for example, and direct interaction with assembly language. These constructs are as effective as those of any high-level programming language (yes, including C). These constructs are expressively powerful, well-specified (so there are few surprises), efficient, and portable (within reason).

We say "within reason" because portability is a difficult goal for embedded systems. That's because the hardware is so much a part of the application itself, rather than being abstracted away as in a general-purpose application. That said, the hardware details can be managed in Ada so that portability is maximized to the extent possible for the application.

But strictly speaking, not all software can or should be absolutely portable! If a specific device is required, well, the program won't work with some other device. But to the extent possible portability is obviously a good thing.

30.4 The Ada Drivers Library

Speaking of SoC computers, there is a library of freely-available device drivers in Ada. Known as the Ada Driver Library (ADL), it supports many devices on a number of vendors' products. Device drivers for timers, I2C, SPI, A/D and D/A converters, DMA, General Purpose I/O, LCD displays, sensors, and other devices are included. The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.

An extensive description of a project using the ADL is available here: <https://blog.adacore.com/making-an-rc-car-with-ada-and-spark>

We will refer to components of this library and use some of them as examples.

LOW LEVEL PROGRAMMING

This section introduces a number of topics in low-level programming, in which the hardware and the compiler's representation choices are much more in view at the source code level. In comparatively high level code these topics are "abstracted away" in that the programmer can assume that the compiler does whatever is necessary on the current target machine so that their code executes as intended. That approach is not sufficient in low-level programming.

Note that we do not cover every possibility or language feature. Instead, we cover the necessary concepts, and also potential surprises or pitfalls, so that the parts not covered can be learned on your own.

31.1 Separation Principle

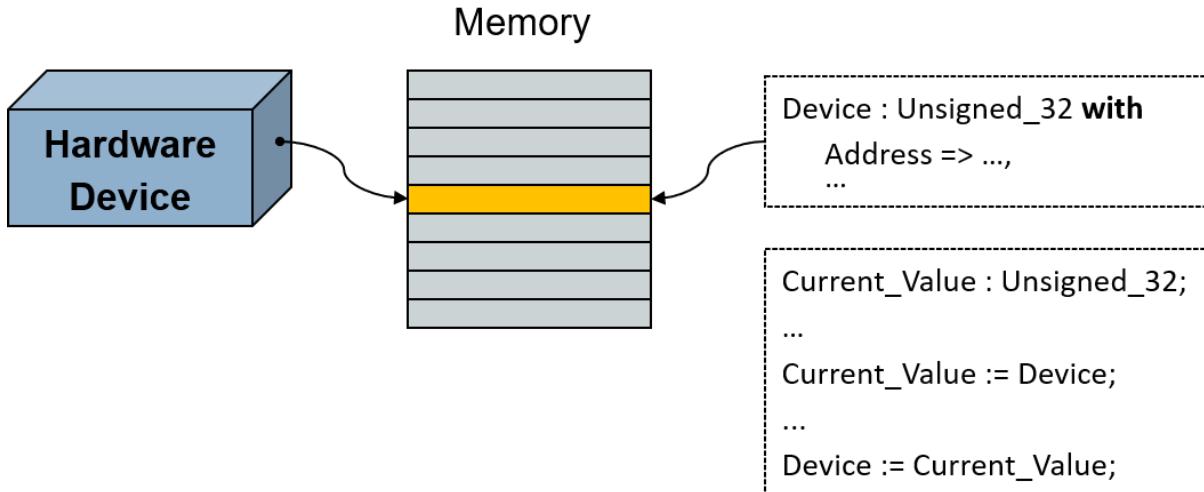
There is a language design principle underlying the Ada facilities intended for implementing embedded software. This design principle directly affects how the language is used, and therefore, the portability and readability of the resulting application code.

This language design principle is known as the "separation principle." What's being separated? The low-level, less portable aspects of some piece of code are separated from the usage of that piece of code.

Don't confuse this with hiding unnecessary implementation details via compile-time visibility control (i.e., information hiding and encapsulation). That certainly should be done too. Instead, because of the separation principle, we specify the low-level properties of something once, when we declare it. From then on, we can use regular Ada code to interact with it. That way the bulk of the code — the usage — is like any other Ada code, and doesn't propagate the low-level details all over the client code. This greatly simplifies usage and understandability as well as easing porting to new hardware-specific aspects. You change things in one place, rather than everywhere.

For example, consider a device mapped to the memory address space of the processor. To interact with the device we interact with one or more memory cells. Reading input from the device amounts to reading the value at the associated memory location. Likewise, sending output to the device amounts to writing to that location.

To represent this device mapping we declare a variable of an appropriate type and specify the starting address the object should occupy. (There are other ways too, but for a single, statically mapped object this is the simplest approach.) We'd want to specify some other characteristics as well, but let's focus on the address.



If the hardware presents an interface consisting of multiple fields within individual memory cells, we can use a record type instead of a single unsigned type representing a single word. Ada allows us to specify the exact record layout, down to the individual bit level, for any types we may need to use for the record components. When we declare the object we use that record type, again specifying the starting address. Then we can just refer to the object's record components as usual, having the compiler compute the address offsets required to access the components representing the individual hardware fields.

Note that we aren't saying that other languages cannot do this too. Many can, using good programming practices. What we're saying is that those practices are designed into the Ada way of doing it.

31.2 Guaranteed Level of Support

The Ada reference manual has an entire section dedicated to low-level programming. That's section 13, "Representation Issues," which provides facilities for developers to query and control aspects of various entities in their code, and for interfacing to hardware. Want to specify the exact layout for a record type's components? Easy, and the compiler will check your layout too. Want to specify the alignment of a type? That's easy too. And that's just the beginning. We'll talk about these facilities as we go, but there's another point to make about this section.

In particular, section 13 includes recommended levels of support to be provided by language implementations, i.e., compilers and other associated tools. Although the word "recommended" is used, the recommendations are meant to be followed.

For example, section 13.3 says that, for some entity named X, "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." So, for example, if the programmer specifies the address for a memory-mapped variable, the compiler cannot ignore that specification and instead, for the sake of performance, represent that variable using a register. The object must be represented as an addressable entity, as requested by the programmer. (Registers are not addressable.)

We mention this because, although the recommended levels of support are intended to be followed, those recommendations become **requirements** if the Systems Programming (SP) Annex is implemented by the vendor. In that case the vendor's implementation of section 13 must support at least the recommended levels. The SP Annex defines additional, optional functionality oriented toward this programming domain; you want it anyway. (Like all the annexes it adds no new syntax.) Almost all vendors, if not literally all, implement the Annex so you can rely on the recommended levels of support.

31.3 Querying Implementation Limits and Characteristics

Sometimes you need to know more about the underlying machine than is typical for general purpose applications. For example, your numerical analysis algorithm might need to know the maximum number of digits of precision that a floating-point number can have on this specific machine. For networking code, you will need to know the "endianness" of the machine so you can know whether to swap the bytes in an Ethernet packet. You'd go look in the `limits.h` file in C implementations, but in Ada we go to a package named `System` to get this information.

Clearly, these implementation values will vary with the hardware, so the package declares constants with implementation-defined values. The names of the constants are what's portable, you can count on them being the same in any Ada implementation.

However, vendors can add implementation-defined declarations to the language-defined content in package `System`. You might require some of those additions, but portability could then suffer when moving to a new vendor's compiler. Try not to use them unless it is unavoidable. Ideally these additions will appear in the private part of the package, so the implementation can use them but application code cannot.

For examples of the useful, language-defined constants, here are those for the numeric limits of an Ada compiler for an Arm 32-bit SoC:

```
Min_Int          : constant := Long_Long_Integer'First;
Max_Int          : constant := Long_Long_Integer'Last;

Max_Binary_Modulus : constant := 2 ** Long_Long_Integer'Size;
Max_Nonbinary_Modulus : constant := 2 ** Integer'Size - 1;

Max_Base_Digits   : constant := Long_Long_Float'Digits;
Max_Digits        : constant := Long_Long_Float'Digits;

Max_Mantissa      : constant := 63;
Fine_Delta         : constant := 2.0 ** (-Max_Mantissa);
```

`Min_Int` and `Max_Int` supply the most-negative and most-positive integer values supported by the machine.

`Max_Binary_Modulus` is the largest power of two allowed as the modulus of a modular type definition.

But a modular type need not be defined in terms of powers of two. An arbitrary modulus is allowed, as long as it is not bigger than the machine can handle. That's specified by `Max_Nonbinary_Modulus`, the largest non-power-of-two value allowed as the modulus of a modular type definition.

`Max_Base_Digits` is the largest value allowed for the requested decimal precision in a floating-point type's definition.

We won't go over all of the above, you get the idea. Let's examine the more important constants.

Two of the most frequently referenced constants in `System` are the following, especially the first. (The values here are again for the Arm 32-bit SoC):

```
Storage_Unit : constant := 8;
Word_Size    : constant := 32;
```

`Storage_Unit` is the number of bits per memory storage element. Storage elements are the components of memory cells, and typically correspond to the individually addressable

memory elements. A "byte" would correspond to a storage element with the above constant value.

Consider a typical idiom for determining the number of whole storage elements an object named X occupies:

```
Units : constant Integer := (X'Size + Storage_Unit - 1) / Storage_Unit;
```

Remember that 'Size returns a value in terms of bits. There are more direct ways to determine that size information but this will serve as an example of the sort of thing you might do with that constant.

A machine "word" is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of some number of storage elements, maybe one but typically more than one. As the unit the machine natively manipulates, words are expected to be independently addressable. (On some machines only words are independently addressable.)

Word_Size is the number of bits in the machine word. On a 32-bit machine we'd expect Word_Size to have a value of 32; on a 64-bit machine it would probably be 64, and so on.

Storage_Unit and Word_Size are obviously related.

Another frequently referenced declaration in package System is that of the type representing memory addresses, along with a constant for the null address designating no storage element.

```
type Address is private;
Null_Address : constant Address;
```

You may be wondering why type **Address** is a private type, since that choice means that we programmers cannot treat it like an ordinary (unsigned) integer value. Portability is of course the issue, because addressing, and thus address representation, varies among computer architectures. Not all architectures have a flat address space directly referenced by numeric values, although that is common. Some are represented by a base address plus an offset, for example. Therefore, the representation for type **Address** is hidden from us, the clients. Consequently we cannot simply treat address values as numeric values. Don't worry, though. The operations we need are provided.

Package System declares these comparison functions, for example:

```
function "<"  (Left, Right : Address) return Boolean;
function "<=" (Left, Right : Address) return Boolean;
function ">"  (Left, Right : Address) return Boolean;
function ">=" (Left, Right : Address) return Boolean;
function "="   (Left, Right : Address) return Boolean;
```

These functions are intrinsic, i.e., built-in, meaning that the compiler generates the code for them directly at the point of calls. There is no actual function body for any of them so there is no performance penalty.

Any private type directly supports the equality function, and consequently the inequality function, as well as assignment. What we don't get here is address arithmetic, again because we don't have a compile-time view of the actual representation. That functionality is provided by package System.Storage_Elements, a child package we will cover later. We should say though, that the need for address arithmetic in Ada is rare, especially compared to C.

Having type **Address** presented as a private type is not, strictly speaking, required by the language. Doing so is a good idea for the reasons given above, and is common among vendors. Not all vendors do, though.

Note that **Address** is the type of the result of the query attribute **Address**.

We mentioned potentially needing to swap bytes in networking communications software, due to the differences in the "endianness" of the machines communicating. That characteristic can be determined via a constant declared in package System as follows:

```
type Bit_Order is (High_Order_First, Low_Order_First);
Default_Bit_Order : constant Bit_Order := implementation-defined;
```

High_Order_First corresponds to "Big Endian" and Low_Order_First to "Little Endian." On a Big Endian machine, bit 0 is the most significant bit. On a Little Endian machine, bit 0 is the least significant bit.

Strictly speaking, this constant gives us the default order for bits within storage elements in record representation clauses, not the order of bytes within words. However, we can usually use it for the byte order too. In particular, if Word_Size is greater than Storage_Unit, a word necessarily consists of multiple storage elements, so the default bit ordering is the same as the ordering of storage elements in a word.

Let's take that example of swapping the bytes in a received Ethernet packet. The "wire" format is Big Endian so if we are running on a Little Endian machine we must swap the bytes received.

Suppose we want to retrieve typed values from a given buffer or bytes. We get the bytes from the buffer into a variable named Value, of the type of interest, and then swap those bytes within Value if necessary.

```
...
begin
  Value := ...  

  if Default_Bit_Order /= High_Order_First then
    -- we're not on a Big Endian machine
    Value := Byte_Swapped (Value);
  end if;
end Retrieve_4_Bytes;
```

We have elided the code that gets the bytes into Value, for the sake of simplicity. How the bytes are actually swapped by function Byte_Swapped is also irrelevant. The point here is the if-statement: the expression compares the Default_Bit_Order constant to High_Order_First to see if this execution is on a Big Endian machine. If not, it swaps the bytes because the incoming bytes are always received in "wire-order," i.e., Big Endian order.

Another important set of declarations in package System define the values for priorities, including interrupt priorities. We will ignore them until we get to the section on interrupt handling.

Finally, and perhaps surprisingly, a few declarations in package System are almost always (if not actually always) ignored.

```
type Name is implementation-defined-enumeration-type;
System_Name : constant Name := implementation-defined;
```

Values of type Name are the names of alternative machine configurations supported by the implementation. System_Name represents the current machine configuration. We've never seen any actual use of this.

Memory_Size is an implementation-defined value that is intended to reflect the memory size of the configuration, in units of storage elements. What the value actually refers to is not specified. Is it the size of the address space, i.e., the amount possible, or is it the amount of physical memory actually on the machine, or what? In any case, the amount of memory available to a given computer is neither dependent upon, nor reflected by, this constant. Consequently, Memory_Size is not useful either.

Why have something defined in the language that nobody uses? In short, it seemed like a good idea at the time when Ada was first defined. Upward-compatibility concerns propagate these declarations forward as the language evolves, just in case somebody does use them.

31.4 Querying Representation Choices

As we mentioned in the introduction, in low-level programming the hardware and the compiler's representation choices can come to the forefront. You can, therefore, query many such choices.

For example, let's say we want to query the addresses of some objects because we are calling the imported C `memcpy` function. That function requires two addresses to be passed to the call: one for the source, and one for the destination. We can use the '`Address`' attribute to get those values.

We will explore importing routines and objects implemented in other languages elsewhere. For now, just understand that we will have an Ada declaration for the imported routine that tells the compiler how it should be called. Let's assume we have an Ada function declared like so:

```
function MemCopy
  (Destination : System.Address;
   Source      : System.Address;
   Length      : Natural)
return Address
with
  Import,
  Convention => C,
  Link_Name => "memcpy",
  Pre  => Source /= Null_Address      and then
           Destination /= Null_Address and then
           not Overlapping (Destination, Source, Length),
  Post => MemCopy'Result = Destination;
-- Copies Length bytes from the object designated by Source to the object
-- designated by Destination.
```

The three aspects that do the importing are specified after the reserved word `with` but can be ignored for this discussion. We'll talk about them later. The preconditions make explicit the otherwise implicit requirements for the arguments passed to `memcpy`, and the postcondition specifies the expected result returned from a successful call. Neither the preconditions nor the postconditions are required for importing external entities but they are good "guard-rails" for using those entities. If we call it incorrectly the precondition will inform us, and likewise, if we misunderstand the result the postcondition will let us know (at least to the extent that the return value does that).

For a sample call to our imported routine, imagine that we have a procedure that copies the bytes of a `String` parameter into a `Buffer` parameter, which is just a contiguous array of bytes. We need to tell `Memcpy` the addresses of the arguments passed so we apply the '`Address`' attribute accordingly:

```
procedure Put (This : in out Buffer; Start : Index; Value : String) is
  Result : System.Address with Unreferenced;
begin
  Result := MemCopy (Destination => This (Start)'Address,
                     Source      => Value'Address,
                     Length      => Value'Length);
end Put;
```

The order of the address parameters is easily confused so we use the named association format for specifying the actual parameters in the call.

Although we assign `Result` we don't otherwise use it, so we tell the compiler this is not a mistake via the `Unreferenced` aspect. And if we do turn around and reference it the compiler will complain, as it should. Note that `Unreferenced` is defined by GNAT, so usage is not necessarily portable. Other vendors may or may not implement something like it, perhaps with a different name.

(We don't show the preconditions for `Put`, but they would have specified that `Start` must be a valid index into this particular buffer, and that there must be room in the `Buffer` argument for the number of bytes in `Value` when starting at the `Start` index, so that we don't copy past the end of the `Buffer` argument.)

There are other characteristics we might want to query too.

We might want to ask the compiler what alignment it chose for a given object (or type, for all such objects).

For a type, when `Alignment` returns a non-zero value we can be sure that the compiler will allocate storage for objects of the type at correspondingly aligned addresses (unless we force it to do otherwise). Similarly, references to dynamically allocated objects of the type will be to properly aligned locations. Otherwise, an `Alignment` of zero means that the guarantee does not hold. That could happen if the type is packed down into a composite object, such as an array of Booleans. We'll discuss "packing" soon. More commonly, the smallest likely value is 1, meaning that any storage element's address will suffice. If the machine has no particular natural alignments, then all type alignments will probably be 1 by default. That would be somewhat rare today, though, because modern processors usually have comparatively strict alignment requirements.

We can ask for the amount of storage associated with various entities. For example, when applied to a task, '`Storage_Size`' tells us the number of storage elements reserved for the task's execution. The value includes the size of the task's stack, if it has one. We aren't told if other required storage, used internally in the implementation, is also included in this number. Often that other storage is not included in this number, but it could be.

`Storage_Size` is also defined for access types. The meaning is a little complicated. Access types can be classified into those that designate only variables and constants ("access-to-object") and those that can designate subprograms. Each access-to-object type has an associated storage pool. The storage allocated by `new` comes from the pool, and instances of `Unchecked_Deallocation` return storage to the pool.

When applied to an access-to-object type, `Storage_Size` gives us the number of storage elements reserved for the corresponding pool.

Note that `Storage_Size` doesn't tell us how much available, unallocated space remains in a pool. It includes both allocated and unallocated space. Note, too, that although each access-to-object type has an associated pool, that doesn't mean that each one has a distinct, dedicated pool. They might all share one, by default. On an operating system, such as Linux, the default shared pool might even be implicit, consisting merely of calls to the OS routines in C.

As a result, querying `Storage_Size` for access types and tasks is not necessarily all that useful. Specifying the sizes, on the other hand, definitely can be useful.

That said, we can create our own pool types and define precisely how they are sized and how allocation and deallocation work, so in that case querying the size for access types could be more useful.

For an array type or object, '`Component_Size`' provides the size in bits of the individual components.

More useful are the following two attributes that query a degree of memory sharing between objects.

Applied to an object, '`Has_Same_Storage`' is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations occupy exactly the same bits.

Applied to an object, '[Overlaps_Storage](#)' is a Boolean function that takes another object of any type as the argument. It indicates whether the two objects' representations share at least one bit.

Generally, though, we specify representation characteristics far more often than we query them. Rather than describe all the possibilities, we can just say that all the representation characteristics that can be specified can also be queried. We cover specifying representation characteristics next, so just assume the corresponding queries are available.

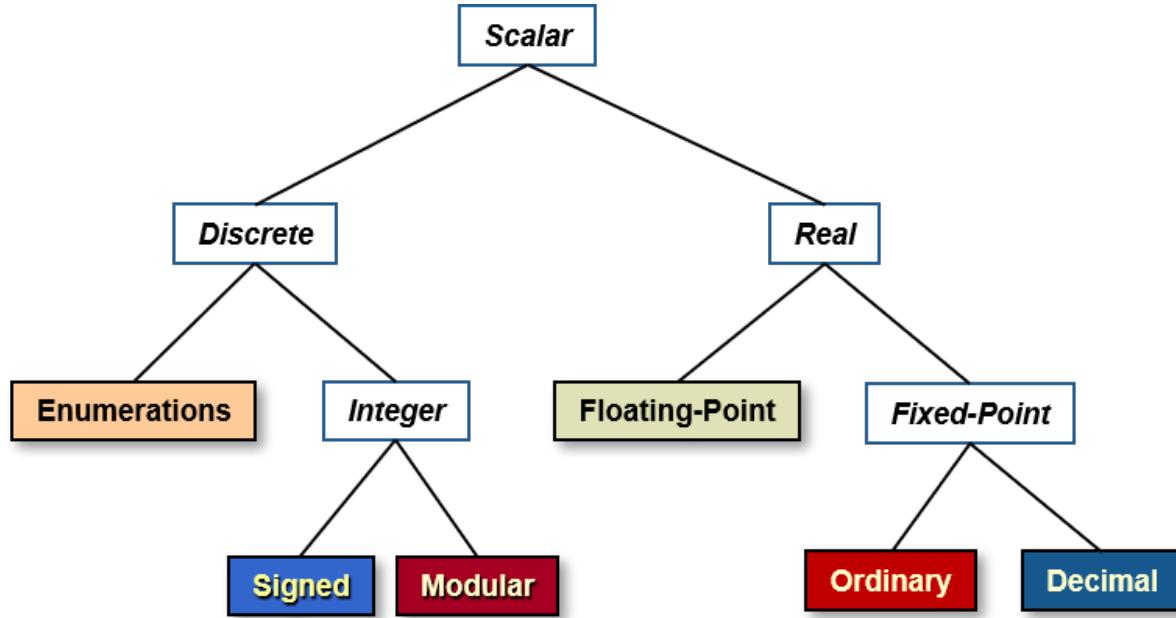
That said, there is one particular representation query we need to talk about explicitly, now, because there is a lot of confusion about it: the '[Size](#)' attribute. The confusion stems from the fact that there are multiple contexts for applying the attribute, and multiple reasonable interpretations possible. We can apply the '[Size](#)' attribute to a type, in an attempt to get information about all objects of the type, or we can apply it to individual objects to get specific information. In both cases, what actual information do we get? In the original version of Ada these questions weren't really answered so vendors did what they thought was correct. But they did not agree with each other, and portability became a problem.

For example, suppose you want to convert some value to a series of bytes in order to send the value over the wire. To do that you need to know how many bytes are required to represent the value. Many applications queried the size of the type to determine that, and then, when porting to a new vendor's compiler, found that their code no longer worked correctly. The new vendor's implementation wasn't wrong, it was just different.

Later versions of Ada answered these questions, where possible, so let's examine the contexts and meaning. Above all, though, remember that '[Size](#)' returns values in terms of **bits**.

If we apply '[Size](#)' to a type, the resulting value depends on the kind of type.

For scalar types, the attribute returns the *minimum* number of bits required to represent all the values of the type. Here's a diagram showing what the category "scalar types" includes:



Consider type **Boolean**, which has two possible values. One bit will suffice, and indeed the language standard requires **Boolean**'[Size](#)' to be the value 1.

This meaning also applies to subtypes, which can constrain the number of values for a scalar type. Consider subtype **Natural**. That's a subtype defined by the language to be type **Integer** but with a range of **0 .. Integer'Last**. On a 32-bit machine we would expect **Integer** to be a native type, and thus 32-bits. On such a machine if we say **Integer**'[Size](#)'

we will indeed get 32. But if we say **Natural'Size** we will get 31, not 32, because only 31 bits are needed to represent that range on that machine.

The size of objects, on the other hand, cannot be just a matter of the possible values. Consider type **Boolean** again, where **Boolean'Size** is required to be 1. No compiler is likely to allocate one bit to a **Boolean** variable, because typical machines don't support individually-addressable bits. Instead, addresses refer to storage elements, of a size indicated by the **Storage_Unit** constant. The compiler will allocate the smallest number of storage elements necessary, consistent with other considerations such as alignment. Therefore, for a machine that has **Storage_Unit** set to a value of eight, we can assume that a compiler for that machine will allocate an entire eight-bit storage element to a stand-alone **Boolean** variable. The other seven bits are simply not used by that variable. Moreover, those seven bits are not used by any other stand-alone object either, because access would be far less efficient, and such sharing would require some kind of locking to prevent tasks from interfering with each other when accessing those stand-alone objects. (Stand-alone objects are independently addressable; they wouldn't stand alone otherwise.)

By the same token (and still assuming a 32-bit machine), a compiler will allocate more than 31 bits to a variable of subtype **Natural** because there is no 31-bit addressable unit. The variable will get all 32-bits.

Note that we're talking about individual, stand-alone variables. Components of composite types, on the other hand, might indeed share bytes if the individual components don't require all the bits of their storage elements. You'd have to request that representation, though, with most implementations, because accessing the components at run-time would require more machine instructions. We'll go into the details of that later.

Let's talk further about sizes of types.

For record types, '**Size**' gives the minimum number of bits required to represent the whole composite value. But again, that's not necessarily the number of bits required for the objects' in-memory representation. The order of the components within the record can make a difference, as well as their alignments. The compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. As a result the overall size could be larger.

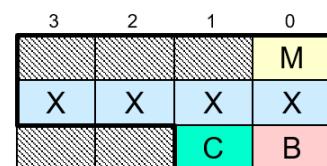
Note that Ada compilers are allowed to reorder the components; the order in memory might not match the order in the source code.

For example, consider this record type and its components:

```
type My_Int is range 1..10;
subtype S is Integer range 1..10;
type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```

If compiler allocates
in declaration order

Sample layout for a given compiler



R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects

In the figure, we see a record type with some components, and a sample layout for that record type assuming the compiler does not reorder the components. Observe that some bytes allocated to objects of type **R** are unused (the darkly shaded ones). In this case that's because the alignment of subtype **S** happens to be 4 on this machine. The component **X** of that subtype **S** cannot start at byte offset 1, or 2, or 3, because those addresses would not satisfy the alignment constraint of **S**. (We're assuming byte 0 is at a word-aligned address.) Therefore, **X** starts at the object's starting address plus 4. Components **B** and **C** are of types

that have an alignment of 1, so they can start at any storage element. They immediately follow the bytes allocated to component X. Therefore, R'Size is 80, or 10 bytes. The three bytes following component M are simply not used.

But what about the two bytes following the last component C? They could be allocated to stand-alone objects if they would fit. More likely, though, the compiler will allocate those two bytes to objects of type R, that is, 12 bytes instead of 10 are allocated. As a result, 96 bits are actually used in memory. The extra, unused 16 bits are "padding."

Why add unused padding? It simplifies the memory allocation of objects of type R. Suppose some array type has components of record type R. Assuming the first component is aligned properly, every following component will also be aligned properly, automatically, because the two padding bytes are considered parts of the components.

To make that work, the compiler takes the most stringent alignment of all the record type's components and uses that for the alignment of the overall record type. That way, any address that satisfies the record object's alignment will satisfy the components' alignment requirements. The alignment is component X, of subtype S, is 4. The other components have an alignment of 1, therefore R'Alignment is 4. An aligned address plus 12 will also be an aligned address.

This rounding up based on alignment is recommended behavior for the compiler, not a requirement, but is reasonable and typical among vendors. Although it can result in unused storage, that's the price paid for speed of access (or even correctness for machines that would fault on misaligned component accesses).

As you can see, alignment is a critical factor in the sizes of composite objects. If you care about the layout of the type you very likely need to care about the alignment of the components and overall record type.

Ada compilers are allowed to reorder the components of record types in order to minimize these gaps or satisfy the alignment requirements of the components. Some compilers do, some don't. Consider the type R again, this time with the first two components switched in the component declaration order:

```
type My_Int is range 1..10;
subtype S is Integer range 1..10;
type R is record
  X : S;
  M : My_Int;
  B : Boolean;
  C : Character;
end record;
```

Sample layout for a given compiler

If compiler allocates
in declaration order

3	2	1	0
X	X	X	X
	C	B	M

R'Size will be 56 bits (7 bytes,
but all 8 will be allocated)

Now R'Size will report 56 bits instead of 80. The one trailing byte will still be padding, but only that one.

What about unbounded types, for example type **String**? Querying the 'Size in that case would provide an implementation-defined result. A somewhat silly thing to do, really, since the type — by definition — doesn't specify how many components are involved.

Usually, though, you don't want to query the size of a type. Most of the time what you want is the size of objects of the type. Going back to sending values over the wire, the code should query the size of the *parameter* holding the value to be sent. That will tell you how many bits are really needed.

One last point: GNAT, and now Ada 202x, define an attribute named **Object_Size**. It does just what the name suggests: what 'Size does when applied to objects rather than types.

GNAT also defines another attribute, named `Value_Size`, that does what '`Size`' does when applied to types. The former is far more useful so Ada has standardized it.

31.5 Specifying Representation

Recall that we said `Boolean'Size` is required to be 1, and that stand-alone objects of type `Boolean` are very likely allocated some integral number of storage elements (e.g., bytes) in memory, typically one. What about arrays of Booleans? Suppose we have an array of 16 Boolean components. How big are objects of the array type? It depends on the machine. Continuing with our hypothetical (but typical) byte-addressable machine, for the sake of efficient access each component is almost certainly allocated an individual byte rather than a single bit, just like stand-alone objects. Consequently, our array of 16 Booleans will be reported by '`Size`' to be 128 bits, i.e., 16 bytes. If you wanted a bit-mask, in which each Boolean component is allocated a single bit and the total array size is 16 bits, you'd have a problem. The compiler assumes you want speed of access rather than storage minimization, and normally that would be the right assumption.

Naturally there is a solution. Ada allows us to specify the representation characteristics of types, and thus objects of those types, including their bit-wise layouts. It also allows us to specify the representation of individual objects. You should understand, though, that the compiler is not required to do what you ask, because you might ask the impossible. For example, if you specify that the array of 16 Booleans is to be represented completely in 15 bits, what can the compiler do? Rejecting that specification is the only reasonable response. But if you specify something possible, the compiler must do what you ask, absent some compelling reason to the contrary.

With that in mind, let's examine setting the size for types.

So, how do we specify that we want our array of 16 Boolean components to be allocated one bit per component, for a total allocation of 16 bits? There are a couple of ways, one somewhat better than the other.

First, you can ask that the compiler "pack" the components into as small a number of bits as it can:

```
type Bits16 is array (0 .. 15) of Boolean with
  Pack;
```

That likely does what you want: `Bits16'Size` will probably be 16.

But realize that the `Pack` aspect (and corresponding pragma) is merely a request that the compiler do its best to minimize the number of bits allocated, not necessarily that it do exactly what you expected or required.

We could set the size of the entire array type:

```
type Bits16 is array (0 .. 15) of Boolean with
  Size => 16;
```

But the language standard says that a `Size` clause on array and record types should not affect the internal layout of their components. That's Implementation Advice, so not normative, but implementations are really expected to follow the advice, absent some compelling reason. That's what the `Pack` aspect, record representation clauses, and `Component_Size` clauses are for. (We'll talk about record representation clauses momentarily.) That said, at least one other vendor's compiler would have changed the size of the array type because of the `Size` clause, so GNAT defines a configuration pragma named `Implicit_Packing` that overrides the default behavior. With that pragma applied, the `Size` clause would compile and suffice to make the overall size be 16. That's a vendor-defined pragma though, so not portable.

Therefore, the best way to set the size for the array type is to set the size of the individual components, via the Component_Size aspect as the Implementation Advice indicates. That will say what we really want, rather than a "best effort" request for the compiler, and is portable:

```
type Bits16 is array (0 .. 15) of Boolean with
    Component_Size => 1;
```

With this approach the compiler must either use the specified size for each component or refuse to compile the code. If it compiles, objects of the array type will be 16 bits total (plus any padding bits required to make objects have a size that is a multiple of Storage_Unit, typically zero on modern machines).

Now that we have a bit-mask array type, let's put it to use.

Let's say that you have an object that is represented as a simple signed integer because, for most usage, that's the appropriate representation. Sometimes, though, let's say you need to access individual bits of the object instead of the whole numeric value. Signed integer types don't provide bit-level access. In Ada we'd say that the "view" presented by the object's type doesn't include bit-oriented operations. Therefore, we need to add a view to the object that does provide them. A different view will require an additional type for the same object.

Applying different types, and thus their operations, to the same object is known as [type punning](#)⁴⁷ in computer programming. Realize that doing so circumvents the static strong typing we harness to protect us from ourselves and from others. Use it with care! (For example, limit the compile-time visibility to such code.)

One way to add a view is to express an "overlay," in which an object of one type is placed at the same memory location as a distinct object of a different type, thus "overlaid" one object over the other in memory. The different types present different views, therefore different operations available for the shared memory cells. Our hypothetical example uses two views, but you can overlay as many different views as needed. (That said, requiring a large number of different views of the same object would be suspect.)

There are other ways in Ada to apply different views, some more flexible than others, but an overlay is a simple one that will often suffice.

Here is an implementation of the overlay approach, using our bit-mask array type:

```
type Bits32 is array (0 .. 31) of Boolean with
    Component_Size => 1;

X : Integer;
Y : Bits32 with Address => X'Address;
```

We can query the addresses of objects, and other things too, but objects, especially variables, are the most common case. In the above, we say X'Address to query the starting address of object X. With that information we know what address to specify for our bit-mask overlay object Y. Now X and Y are aliases for the same memory cells, and therefore we can manipulate and query that memory as either a signed integer or as an array of bits. Reading or updating individual array components accesses the individual bits of the overlaid object.

Instead of the Bits32 array type, we could have specified a modular type for the overlay Y to get a view providing bit-oriented operations. Overlaying such an array was a common idiom prior to the introduction of modular "unsigned" types in Ada, and remains useful for accessing individual bits. In other words, using a modular type for Y, you could indeed access an individual bit by passing a mask value to the **and** operator defined in any modular type's view. Using a bit array representation lets the compiler do that work for you, in the generated code. The source code will be both easier to read and more explicit about what it is doing when using the bit array overlay.

⁴⁷ https://en.wikipedia.org/wiki>Type_punning

One final issue remains: in our specific overlay example the compiler would likely generate code that works. But strictly speaking it might not.

The Ada language rules say that for such an overlaid object — Y in the example above — the compiler should not perform optimizations regarding Y that it would otherwise apply in the absence of aliases. That's necessary, functionally, but may imply degraded performance regarding Y, so keep it in mind. Aliasing precludes some desirable optimizations.

But what about X in the example above? We're querying that object's address, not specifying it, so the RM rule precluding optimizations doesn't apply to X. That can be problematic.

The compiler might very well place X in a register, for example, for the sake of the significant performance increase (another way of being friendly). But in that case `System.Null_Address` will be returned by the `X'Address` query and, consequently, the declaration for Y will not result in the desired overlaying.

Therefore, we should mark X as explicitly `aliased` to ensure that `X'Address` is well-defined:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

The only difference in the version above is the addition of `aliased` in the declaration of X. Now we can be certain that the optimizer will not represent X in some way incompatible with the idiom, and `X'Address` will be well-defined.

In our example X and Y are clearly declared in the same compilation unit. Most compilers will be friendly in this scenario, representing X in such a way that querying the address will return a non-null address value even if `aliased` is not applied. Indeed, `aliased` is relatively new to Ada, and earlier compilers typically emitted code that would handle the overlay as intended.

But suppose, instead of being declared in the same declarative part, that X was declared in some other compilation unit. Let's say it is in the visible part of a package declaration. (Assume X is visible to clients for some good reason.) That package declaration can be, and usually will be, compiled independently of clients, with the result that X might be represented in some way that cannot support querying the address meaningfully.

Therefore, the declaration of X in the package spec should be marked as aliased, explicitly:

```
package P is
  X : aliased Integer;
end P;
```

Then, in the client code declaring the overlay, we only declare Y, assuming a with-clause for P:

```
type Bits32 is array (0 .. 31) of Boolean with
  Component_Size => 1;

Y : Bits32 with Address => P.X'Address;
```

All well and good, but how did the developer of the package know that some other unit, a client of the package, would query the address of X, such that it needed to be marked as aliased? Indeed, the package developer might not know. Yet the programmer is responsible for ensuring a valid and appropriate `Address` value is used in the declaration of Y. Execution is erroneous otherwise, so we can't say what would happen in that case. Maybe an exception is raised or a machine trap, maybe not.

Worse, the switches that were applied when compiling the spec for package P can make a difference: P.X might not be placed in a register unless the optimizer is enabled. Hence the client code using Y might work as expected when built for debugging, with the optimizer

disabled, and then not do so when re-built for the final release. You'd probably have to solve this issue by debugging the application.

On a related note, you may be asking yourself how to know that type **Integer** is 32 bits wide, so that we know what size array to use for the bit-mask. The answer is that you just have to know the target well when doing low-level programming. The hardware becomes much more visible, as we mentioned.

That said, you could at least verify the assumption:

```
pragma Compile_Time_Error (Integer'Object_Size /= 32,
                           "Integers expected to be 32 bits");
X : aliased Integer;
Y : Bits32 with Address => X'Address;
```

That's a vendor-defined pragma so this is not fully portable. It isn't an unusual pragma, though, so at least you can probably get the same functionality even if the pragma name varies.

Overlays aren't always structured like our example above, i.e., with two objects declared at the same time. We might apply a different type to the same memory locations at different times. Here's an example from the ADL to illustrate the idea. We'll elaborate on this example later, in another section.

First, a package declaration, with two functions that provide a device-specific unique identifier located in shared memory. Each function provides the same Id value in a distinct format. One format is a string of 12 characters, the other is a sequence of three 32-bit values. Hence both representations are the same size.

```
package STM32.Device_Id is

  subtype Device_Id_Image is String (1 .. 12);

  function Unique_Id return Device_Id_Image;

  type Device_Id_Tuple is array (1 .. 3) of UInt32
    with Component_Size => 32;

  function Unique_Id return Device_Id_Tuple;

end STM32.Device_Id;
```

In the package body we implement the functions as two ways to access the same shared memory, specified by ID_Address:

```
with System;

package body STM32.Device_Id is

  ID_Address : constant System.Address := System'To_Address (16#1FFF_7A10#);

  function Unique_Id return Device_Id_Image is
    Result : Device_Id_Image with Address => ID_Address, Import;
  begin
    return Result;
  end Unique_Id;

  function Unique_Id return Device_Id_Tuple is
    Result : Device_Id_Tuple with Address => ID_Address, Import;
  begin
    return Result;
  end Unique_Id;
```

(continues on next page)

(continued from previous page)

```
end STM32.Device_Id;
```

`System'To_Address` is just a convenient way to convert a numeric value into an `Address` value. The primary benefit is that the call is a static expression, but we can ignore that here. Using `Import` is a good idea to ensure that the Ada code does no initialization of the object, since the value is coming from the hardware via the shared memory. Doing so may not be necessary, depending on the type used, but is a good habit to develop.

The point of this example is that we have one object declaration per function, of a type corresponding to the intended function result type. Because each function places their local object at the same address, they are still overlaying the shared memory.

Now let's return, momentarily, to setting the size of entities, but now let's focus on setting the size of objects.

We've said that the size of an object is not necessarily the same as the size of the object's type. The object size won't be smaller, but it could be larger. Why? For a stand-alone object or a parameter, most implementations will round the size up to a storage element boundary, or more, so the object size might be greater than that of the type. Think back to `Boolean`, where `Size` is required to be 1, but stand-alone objects are probably allocated 8 bits, i.e., an entire storage element (on our hypothetical byte-addressed machine).

Likewise, recall that numeric type declarations are mapped to underlying hardware numeric types. These underlying numeric types provide at least the capabilities we request with our type declarations, e.g., the range or number of digits, perhaps more. But the mapped numeric hardware type cannot provide less than requested. If there is no underlying hardware type with at least our requested capabilities, our declarations won't compile. That mapping means that specifying the size of a numeric type doesn't necessarily affect the size of objects of the type. That numeric hardware type is the size that it is, and is fixed by the hardware.

For example, let's say we have this declaration:

```
type Device_Register is range 0 .. 2**5 - 1 with Size => 5;
```

That will compile successfully, because there will be a signed integer hardware type with at least that range. (Not necessarily, legally speaking, but realistically speaking, there will be such a hardware type.) Indeed, it may be an 8-bit signed integer, in which case `Device_Register'Size` will give us 5, but objects of the type will have a size of 8, unavoidably, even though we set `Size` to 5.

The difference between the type and object sizes can lead to potentially problematic code:

```
type Device_Register is range 0 .. 2**8 - 1 with Size => 8;
```

```
My_Device : Device_Register
  with Address => To_Address (...);
```

The code compiles successfully, and tries to map a byte to a hardware device that is physically connected to one storage element in the processor memory space. The actual address is elided as it is not important here.

That code might work too, but it might not. We might think that `My_Device'Size` is 8, and that `My_Device'Address` points at an 8-bit location. However, this isn't necessarily so, as we saw with the supposedly 5-bit example earlier. Maybe the smallest signed integer the hardware has is 16-bits wide. The code would compile because a 16-bit signed numeric type can certainly handle the 8-bit range requested. `My_Device'Size` would be then 16, and because '`Address`' gives us the *starting* storage element, `My_Device'Address` might designate the high-order byte of the overall 16-bit object. When the compiler reads the two bytes for `My_Device` what will happen? One of the bytes will be the data presented by

the hardware device mapped to the memory. The other byte will contain undefined junk, whatever happens to be in the memory cell at the time. We might have to debug the code a long time to identify that as the problem. More likely we'll conclude we have a failed device.

The correct way to write the code is to specify the size of the object instead of the type:

```
type Device_Register is range 0 .. 2**8 - 1;  
  
My_Device : Device_Register with  
  Size => 8,  
  Address => To_Address (...);
```

If the compiler cannot support stand-alone 8-bit objects for the type, the code won't compile.

Alternatively, we could change the earlier `Size` clause on the type to apply `Object_Size` instead:

```
type Device_Register is range 0 .. 2**8 - 1 with Object_Size => 8;  
  
My_Device : Device_Register with  
  Address => To_Address (...);
```

The choice between the two approaches comes down to personal preference, at least if only a small number of stand-alone objects of the type are going to be declared. With either approach, if the implementation cannot support 8-bit stand-alone objects, we find out that there is a problem at compile-time. That's always cheaper than debugging.

You might conclude that setting the `Size` for a type serves no purpose. That's not an unreasonable conclusion, given what you've seen, but in fact there are reasons to do so. However, there are only a few specific cases so we will save the reasons for the discussions of the specific cases.

There is one general case, though, for setting the '`Size`' of a type. Specifically, you may want to specify the size that you think is the minimum possible, and you want the compiler to confirm that belief. This would be one of the so-called "confirming" representation clauses, in which the representation detail is what the compiler would have chosen anyway, absent the specification. You're not actually changing anything, you're just getting confirmation via `Size` whether or not the compiler accepts the clause. Suppose, for example, that you have an enumeration type with 256 values. For enumeration types, the compiler allocates the smallest number of bits required to represent all the values, rounded up to the nearest storage element. (It's not like C, where enums are just named int values.) For 256 values, an eight-bit byte would suffice, so setting the size to 8 would be confirming. But suppose we actually had 257 enumerals, accidentally? Our size clause set to 8 would not compile, and we'd be told that something is amiss.

However, note that if your supposedly "confirming" size clause actually specifies a size larger than what the compiler would have chosen, you won't know, because the compiler will silently accept sizes larger than necessary. It just won't accept sizes that are too small.

There are other confirming representation clauses as well. Thinking again of enumeration types, the underlying numeric values are integers, starting with zero and consecutively increasing from there up to $N-1$, where N is the total number of enumeral values.

For example:

```
type Commands is (Off, On);  
  
for Commands use (Off => 0, On => 1);
```

As a result, `Off` is encoded as 0 and `On` as 1. That specific underlying encoding is guaranteed by the language, as of Ada 95, so this is just a confirming representation clause nowadays.

But it was not guaranteed in the original version of the language, so if you wanted to be sure of the encoding values you would have specified the above. It wasn't necessarily confirming before Ada 95, in other words.

But let's also say that the underlying numeric values are not what you want because you're interacting with some device and the commands are encoded with values other than 0 and 1. Maybe you want to use an enumeration type because you want to specify all the possible values actually used by clients. If you just used some numeric type instead and made up constants for On and Off, there's nothing to keep clients from using other numeric values in place of the two constants (absent some comparatively heavy code to prevent that from happening). Better to use the compiler to make that impossible in the first place, rather than debug the code to find the incorrect values used. Therefore, we could specify different encodings:

```
for Commands use (Off => 2, On => 4);
```

Now the compiler will use those encoding values instead of 0 and 1, transparently to client code.

The encoding values specified must maintain the relative ordering, otherwise the relational operators won't work correctly. For example, for type Commands above, Off is less than On, so the specified encoding value for Off must be less than that of On.

Note that the values given in the example no longer increase consecutively, i.e., there's a gap. That gap is OK, in itself. As long as we use the two enumerals the same way we'd use named constants, all is well. Otherwise, there is both a storage issue and a performance issue possible. Let's say that we use that enumeration type as the index for an array type. Perfectly legal, but how much storage is allocated to objects of this array type? Enough for exactly two components? Four, with two unused? The answer depends on the compiler, and is therefore not portable. The bigger the gaps, the bigger the overall storage difference possible. Likewise, imagine we have a for-loop iterating over the index values of one of these array objects. The for-loop parameter cannot be coded by the compiler to start at 0, clearly, because there is no index (enumeration) value corresponding to 0. Similarly, to get the next index, the compiler cannot have the code simply increment the current value. Working around that takes some extra code, and takes some extra time that would not be required if we did not have the gaps.

The performance degradation can be significant compared to the usual code generated for a for-loop. Some coding guidelines say that you shouldn't use an enumeration representation clause for this reason, with or without gaps. Now that Ada has type predicates we could limit the values used by clients for a numeric type, so an enumeration type is not the only way to get a restricted set of named, encoded values.

```
type Commands is new Integer with
  Static_Predicate => Commands in 2 | 4;

On    : constant Commands := 2;
Off   : constant Commands := 4;
```

The storage and performance issues bring us back to confirming clauses. We want the compiler to recognize them as such, so that it can generate the usual code, thereby avoiding the unnecessary portability and performance issues. Why would we have such a confirming clause now? It might be left over from the original version of the language, written before the Ada 95 change. Some projects have lifetimes of several decades, after all, and changing the code can be expensive (certified code, for example). Whether the compiler does recognize confirming clauses is a feature of the compiler implementation. We can expect a mature compiler to do so, but there's no guarantee.

Now let's turn to what is arguably the most common representation specification, that of record type layouts.

Recall from the discussion above that Ada compilers are allowed to reorder record com-

ponents in physical memory. In other words, the textual order in the source code is not necessarily the physical order in memory. That's different from, say, C, where what you write is what you get, and you better know what you're doing. On some targets a misaligned **struct** component access will perform very poorly, or even trap and halt, but that's not the C compiler's fault. In Ada you'd have to explicitly specify the problematic layout. Otherwise, if compilation is successful, the Ada compiler must find a representation that will work, either by reordering the components or by some other means. Otherwise it won't compile.

GNAT did not reorder components until relatively recently but does now, at least for the more egregious performance cases. It does this reordering silently, too, although there is a switch to have it warn you when it does. To prevent reordering, GNAT defines a pragma named `No_Component_Reorder` that does what the name suggests. You can apply it to individual record types, or globally, as a configuration pragma. But of course because the pragma is vendor defined it is not portable.

Therefore, if you care about the record components' layout in memory, the best approach is to specify the layout explicitly. For example, perhaps you are passing data to code written in C. In that case, you need the component order in memory to match the order given in the corresponding C struct declaration. That order in memory is not necessarily guaranteed from the order in the Ada source code. The Ada compiler is allowed to chose the representation unless you specify it, and it might chose a different layout from the one given. (Ordinarily, letting the compiler chose the layout is the most desirable approach, but in this case we have an external layout requirement.)

Fortunately, specifying a record type's layout is straightforward. The record layout specification consists of the storage places for some or all components, specified with a record representation clause. This clause specifies the order, position, and size of components (including discriminants, if any).

The approach is to first define the record type, as usual, using any component order you like — you're about to specify the physical layout explicitly, in the next step.

Let's reuse that record type from the earlier discussion:

```
type My_Int is range 1 .. 10;

subtype S is Integer range 1 .. 10;

type R is record
    M : My_Int;
    X : S;
    B : Boolean;
    C : Character;
end record;
```

The resulting layout might be like so, assuming the compiler doesn't reorder the components:

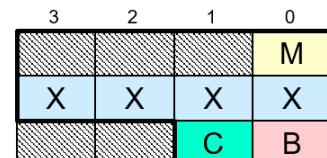
```
type My_Int is range 1..10;

subtype S is Integer range 1..10;

type R is record
    M : My_Int;
    X : S;
    B : Boolean;
    C : Character;
end record;
```

Sample layout for a given compiler

If compiler allocates
in declaration order



R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects

As a result, R'Size will be 80 bits (10 bytes), but those last two bytes will be allocated to objects, for an Object_Size of 96 bits (12 bytes). We'll change that with an explicit layout specification.

Having declared the record type, the second step consists of defining the corresponding record representation clause giving the components' layout. The clause uses syntax that somewhat mirrors that of a record type declaration. The components' names appear, as in a record type declaration. But now, we don't repeat the components' types, instead we give their relative positions within the record, in terms of a relative offset that starts at zero. We also specify the bits we want them to occupy within the storage elements starting at that offset.

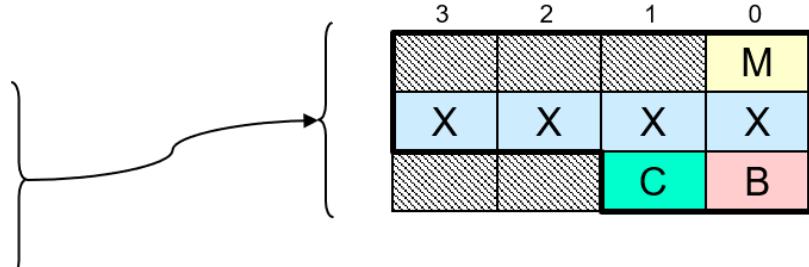
```
for R use record
  X at 0 range 0 .. 31;  -- note the order swap,
  M at 4 range 0 .. 7;  -- with this component
  B at 5 range 0 .. 7;
  C at 6 range 0 .. 7;
end record;
```

Now we'll get the optimized order, and we'll always get that order, or the layout specification won't compile in the first place. In the following diagram, both layouts, the default, and the one resulting from the record representation clause, are depicted for comparison:

```
type My_Int is range 1..10;
```

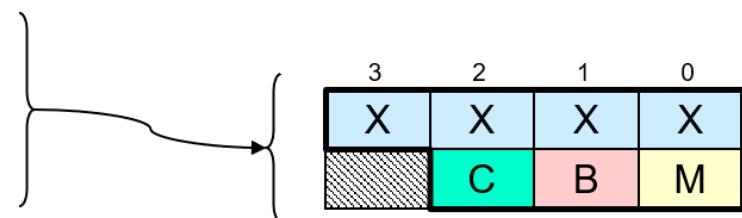
```
subtype S is Integer range 1..10;
```

```
type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```



```
for R use record
```

```
  X at 0 range 0 .. 31;
  M at 4 range 0 .. 7;
  B at 5 range 0 .. 7;
  C at 6 range 0 .. 7;
end record;
```



R'Size will be 56 bits (7 bytes), but that last padding byte will also be allocated to objects, so the Object_Size will be 64 bits (8 bytes).

Notice how we gave each component an offset, after the reserved word **at**. These offsets are in terms of storage elements, and specify their positions within the record object as a whole. They are relative to the beginning of the memory allocated to the record object so they are numbered starting at zero. We want the X component to be the very first component in the allocated memory so the offset for that one is zero. The M component, in comparison, starts at an offset of 4 because we are allocating 4 bytes to the prior component X: bytes 0 through 3 specifically. M just occupies one storage element so the next

component, B, starts at offset 5. Likewise, component C starts at offset 6.

Note that there is no requirement for the components in the record representation clause to be in any particular textual order. The offsets alone specify the components' order in memory. A good style, though, is to order the components in the representation clause so that their textual order corresponds to their order in memory. Doing so facilitates our verifying that the layout is correct because the offsets will be increasing as we read the specification.

An individual component may occupy part of a single storage element, all of a single storage element, multiple contiguous storage elements, or a combination of those (i.e., some number of whole storage elements but also part of another). The bit "range" specifies this bit-specific layout, per component, by specifying the first and last bits occupied. The X component occupies 4 complete 8-bit storage elements, so the bit range is 0 through 31, for a total of 32 bits. All the other components each occupy an entire single storage element so their bit ranges are 0 through 7, for a total of 8 bits.

The text specifying the offset and bit range is known as a "component_clause" in the syntax productions. Not all components need be specified by component_clauses, but (not surprisingly) at most one clause is allowed per component. Really none are required but it would be strange not to have some. Typically, all the components are given positions. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation.

Components not otherwise given an explicit placement are given positions chosen by the compiler. We don't say that they "follow" those explicitly positioned because there's no requirement that the explicit positions start at offset 0, although it would be unusual not to start there.

Placements must not make components overlap, except for components of variant parts, a topic covered elsewhere. You can also specify the placement of implementation-defined components, as long as you have a name to refer to them. (In addition to the components listed in the source code, the implementation can add components to help implement what you wrote explicitly.) Such names are always attribute references but the specific attributes, if any, are implementation-defined. It would be a mistake for the compiler to define such implicit components without giving you a way to refer to them. Otherwise they might go exactly where you want some other component to be placed, or overlap that place.

The positions (offsets) and the bit numbers must be static, informally meaning that they are known at compile-time. They don't have to be numeric literals, though. Numeric constants would work, but literals are the most common by far.

Note that the language does not limit support for component clauses to specific component types. They need not be one of the integer types, in particular. For example, a position can be given for components that are themselves record types, or array types. Even task types are allowed as far as the language goes, although the implementation might require a specific representation, such as the component taking no bits whatsoever (`0 .. -1`). There are restrictions that keep things sane, for example rules about how a component name can be used within the overall record layout construct, but not restrictions on the types allowed for individual components. For example, here is a record layout containing a **String** component, arbitrarily set to contain 11 characters:

```
type R is record
  S : String (1 .. 11);
  B : Boolean;
end record;

for R use record
  S at 0 range 0 .. 87;
  B at 11 range 0 .. 7;
end record;
```

Component S is to be the first component in memory in this example, hence the position offset is 0, for the first byte of S. Next, S is 11 characters long, or 88 bits, so the bit range is 0 .. 87. That's 11 bytes of course, so S occupies storage elements 0 .. 10. Therefore, the next component position must be at least 11, unless there is to be a gap, in which case it would be greater than 11. We'll place B immediately after the last character of S, so B is at storage element offset 11 and occupying all that one byte's bits.

We'll have more to say about record type layouts but first we need to talk about alignment.

Modern target architectures are comparatively strict about the address alignments for some of their types. If the alignment is off, an access to the memory for objects of the type can have highly undesirable consequences. Some targets will experience seriously degraded performance. On others, the target will halt altogether. As you can see, getting the alignment correct is a low-level, but vital, part of correct code on these machines.

Normally the compiler does this work for us, choosing an alignment that is both possible for the target and also optimal for speed of access. You can, however, override the compiler's alignment choice using an attribute definition clause or the Alignment aspect. You can do so on types other than record types, but specifying it on record types is typical. Here's our example record type with the alignment specified via the aspect:

```
type My_Int is range 1 .. 10;

subtype S is Integer range 1 .. 10;

type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record with
  Alignment => 1;
```

Alignment values are in terms of storage elements. The effect of the aspect or attribute clause is to ensure that the starting address of the memory allocated to objects of the type will be a multiple of the specified value.

In fact, whenever we specify a record type layout we really should also specify the record type's alignment, even though doing so is optional. Why? The alignment makes a difference in the overall record object's size. We've seen that already, with the padding bytes: the compiler will respect the alignment requirements of the components, and may add padding bytes within the record and also at the end to ensure components start at addresses compatible with their alignment requirements. The alignment also affects the size allocated to the record type even when the components are already aligned. As a result the overall size could be larger than we want for the sake of space. Additionally, when we pass such objects to code written in other languages, we want to ensure that the starting address of these objects is aligned as the external code expects. The compiler might not choose that required alignment by default.

Specifying alignment for record types is so useful that in the first version of Ada there was no syntax to specify alignment for anything other than record types (via the obsolete **at mod** clause on record representation clauses).

For that reason GNAT provides a pragma named `Optimize_Alignment`. This is a configuration pragma that affects the compiler's choice of default alignments where no alignment is explicitly specified. There is a time/space trade-off in the selection of these values, as we've seen. The normal choice tries to balance these two characteristics, but with an argument to the pragma you can give more weight to one or the other. The best approach is to specify the alignments explicitly, per type, for those that require specific alignment values. The pragma has the nice property of giving general guidance to the compiler for what should be done for the other types and objects not explicitly specified.

Now let's look into the details. We'll use a case study for this purpose, including specifying sizes as well as alignments.

The code for the case study is as follows. It uses Size clauses to specify the Sizes, instead of the Size aspect, just to emphasize that the Size clause approach is not obsolete.

```
package Some_Types is

    type Temperature is range -275 .. 1_000;

    type Identity is range 1 .. 127;

    type Info is record
        T : Temperature;
        Id : Identity;
    end record;

    for Info use record
        T at 0 range 0 .. 15;
        Id at 2 range 0 .. 7;
    end record;

    for Info'Size use 24;

    type List is array (1 .. 3) of Info;
    for List'Size use 24 * 3;

end Some_Types;
```

When we compile this, the compiler will complain that the size for List is too small, i.e., that the minimum allowed is 96 bits instead of the 72 we specified. We specified $24 * 3$ because we said the record size should be 24 bits, and we want our array to contain 3 record components of that size, so 72 seems right.

What's wrong? As we've shown earlier, specifying the record type size doesn't necessarily mean that objects (in this case array components) are that size. The object size could be bigger than we specified for the type. In this case, the compiler says we need 96 total bits for the array type, meaning that each of the 3 array components is 32 bits wide instead of 24.

Why is it 32 bits? Because the alignment for Info is 2 (on this machine). The record alignment is a multiple of the largest alignment of the enclosed components. The alignment for type Temperature (2), is larger than the alignment for type Identity (1), therefore the alignment for the whole record type is 2. We need to go from that number of storage elements to a number of bits for the size.

Here's where it gets subtle. The alignment is in terms of storage elements. Each storage element is of a size in bits given by System.Storage_Unit. We've said that on our hypothetical machine Storage_Unit is 8, so storage elements are 8 bits wide on this machine. Bytes, in other words. Therefore, to get the required size in bits, we have to find a multiple of the two 8-bit bytes (specified by the alignment) that has at least the number of bits we gave in the Size clause. Two bytes only provides 16 bits, so that's not big enough, we need at least 24 bits. The next multiple of 2 bytes is 4 bytes, providing 32 bits, which is indeed larger than 24. Therefore, the overall size of the record type, consistent with the alignment, is 4 bytes, or 32 bits. That's why the compiler says each array component is 32 bits wide.

But for our example let's say that we really want to use only 72 total bits for the array type (and that we want three array components). That's the size we specified, after all. So how do we get the record type to be 24 bits instead of 32? Yes, you guessed it, we change the alignment for the record type. If we change it from 2 to 1, the size of 24 bits will work. Adding this Alignment clause line will do that:

```
for Info'Alignment use 1;
```

An alignment of 1 means that any address will work, assuming that addresses refer to entire storage elements. (An alignment of 0 would mean that the address need not start on a storage element boundary, but we know of no such machines.)

We can even entirely replace the Size clause with the Alignment clause, because the Size clause specifying 24 bits is just confirming: it's the value that 'Size would return anyway. The problem is the object size.

Now, you may be wondering why an alignment of 1 would work, given that the alignment of the Temperature component is 2. Wouldn't it slow down the code, or even trap? Well, maybe. It depends on the machine. If it doesn't work we would just have to use 32 bits for the record type, with the original alignment of 2, for a larger total array size. Of course, if the compiler recognizes that a representation cannot be supported it must reject the code, but the compiler might not recognize the problem.

We said earlier that there are only a small number of reasons to specify 'Size for a type. We can mention one of them now. Setting 'Size can be useful to give the minimum number of bits to use for a component of a packed composite type, that is, within either a record type or an array type that is explicitly packed via the aspect or pragma Pack. It says that the compiler, when giving its best effort, shouldn't compress components of the type any smaller than the number of bits specified. No, it isn't earth-shattering, but other uses are more valuable, to be discussed soon.

One thing we will leave unaddressed (pun intended) is the question of bit ordering and byte ordering within our record layouts. In other words, the "endian-ness". That's a subject beyond the scope of this course. Suffice it to say that GNAT provides a way to specify record layouts that are independent of the endian-ness of the machine, within some implementation-oriented limits. That's obviously useful when the code might be compiled for a different ISA in the future. On the other hand, if your code is specifically for a single ISA, e.g. Arm, even if different boards and hardware vendors are involved, there's no need to be independent of the endian-ness. It will always be the same in that case. (Those are "famous last words" though.) For an overview of the GNAT facility, an attribute named attribute Scalar_Storage_Order see <https://www.adacore.com/papers/lady-ada-mediates-peace-treaty-in-endianness-war>.

Although specifying record type layouts and alignments are perhaps the most common representation characteristics expressed, there are a couple of other useful cases. Both involve storage allocation.

One useful scenario concerns tasking. We can specify the number of storage elements reserved for the execution of a task object, or all objects of a task type. You use the Storage_Size aspect to do so:

```
task Servo with
  Storage_Size => 1 * 1024,
  ...
```

Or the corresponding pragma:

```
task Servo is
  pragma Storage_Size (1 * 1024);
end Servo;
```

The aspect seems textually cleaner and lighter unless you have task entries to declare as well. In that case the line for the pragma wouldn't add all that much. That's a matter of personal aesthetics anyway.

The specified number of storage elements includes the size of the task's stack (GNAT does have one, per task). The language does not specify whether or not it includes other storage associated with the task used for implementing and managing the task execution. With

GNAT, the extent of the primary stack size is the value returned, ignoring any other storage used internally in the run-time library for managing the task.

The GNAT run-time library allocates a default stack amount to each task, with different defaults depending on the underlying O.S., or lack thereof, and the target. You need to read the documentation to find the actual amount, or, with GNAT, read the code.

You would need to specify this amount in order to either increase or decrease the allocated storage. If the task won't run properly, perhaps crashing at strange and seemingly random places, there's a decent chance it is running out of stack space. That might also be the reason if you have a really deep series of subprogram calls that fails. The correction is to increase the allocation, as shown above. How much? Depends on the application code. The quick-and-dirty approach is to iteratively increase the allocation until the task runs properly. Then, reverse the approach until it starts to fail again. Add a little back until it runs, and leave it there. We'll mention a much better approach momentarily (**GNATstack**).

Even if the task doesn't seem to run out of task stack, you might want to reduce it anyway, to the extent possible, because the total amount of storage on your target might be limited. Some of the GNAT bare-metal embedded targets have very small amounts of memory available, so much so that the default task stack allocations would exhaust the memory available quickly. That's what the example above does: empirical data showed that the Servo task could run with just 1K bytes allocated, so we reduced it from the default accordingly. (We specified the size with that expression for the sake of readability, relative to using literals directly.)

Notice we said "empirical data" above. How do we know that we exercised the task's thread of control exhaustively, such that the arrived-at allocation value covers the worst case? We don't, not with certainty. If we really must know the allocation will suffice for all cases, say because this is a high-integrity application, we would use **GNATstack**. GNATstack is an offline tool that exploits data generated by the compiler to compute worst-case stack requirements per subprogram and per task. As a static analysis tool, its computation is based on information known at compile time. It does not rely on empirical run-time information.

The other useful scenario for allocating storage concerns access types, specifically access types whose values designate objects, as opposed to designating subprograms. (Remember, objects are either variables or constants.) There is no notion of dynamically allocating procedures and functions in Ada so access-to-subprogram types are not relevant here. But objects can be of protected types (or task types), and protected objects can "contain" entries and protected subprograms, so there's a lot of expressive power available. You just don't dynamically allocate procedures or functions as such.

First, a little background on access types, to supplement what we said earlier.

By default, the implementation chooses a standard storage pool for each named access-to-object type. The storage allocated by an allocator (i.e., `new`) for such a type comes from the associated pool.

Several access types can share the same pool. By default, the implementation might choose to have a single global storage pool, used by all such access types. This global pool might consist merely of calls to operating system routines (e.g., `malloc`), or it might be a vendor-defined pool instead. Alternatively, the implementation might choose to create a new pool for each access-to-object type, reclaiming the pool's memory when the access type goes out of scope (if ever). Other schemes are possible.

Finally, users may define new pool types, and may override the choice of pool for an access-to-object type by specifying `Storage_Pool` for the type. In this case, allocation (via `new`) takes memory from the user-defined pool and deallocation puts it back into that pool, transparently.

With that said, here's how to specify the storage to be used for an access-to-object type. There are two ways to do it.

If you specify `Storage_Pool` for an access type, you indicate a specific pool object to be used (user-defined or vendor-defined). The pool object determines how much storage is

available for allocation via `new` for that access type.

Alternatively, you can specify `Storage_Size` for the access type. In this case, an implementation-defined pool is used for the access type, and the storage available is at least the amount requested, maybe more (it might round up to some advantageous block size, for example). If the implementation cannot satisfy the request, `Storage_Error` is raised.

It should be clear that the two alternatives are mutually exclusive. Therefore the compiler will not allow you to specify both.

Each alternative has advantages. If your only concern is the total number of allocations possible, use `Storage_Size` and let the implementation do the rest. However, maybe you also care about the behavior of the allocation and deallocation routines themselves, beyond just providing and reclaiming the storage. In that case, use `Storage_Pool` and specify a pool object of the appropriate type. For example, you (or the vendor, or someone else) might create a pool type in which the allocation routine performs in constant time, because you want to do `new` in a real-time application where predictability is essential.

Lastly, an idiom: when using `Storage_Size` you may want to specify a value of zero. That means you intend to do no allocations whatsoever, and want the compiler to reject the code if you try. Why would you want an access type that doesn't allow dynamically allocating objects? It isn't as unreasonable as it might sound. If you plan to use the access type strictly with aliased objects, never doing any allocations, you can have the compiler enforce your intent. There are application domains that prohibit dynamic allocations due to the difficulties in analyzing their behavior, including issues of fragmentation and exhaustion. Access types themselves are allowed in these domains. You'd simply use them to designate aliased objects alone. In addition, in this usage scenario, if the implementation associates an actual pool with each access type, the pool's storage would be wasted since you never intend to allocate any storage from it. Specifying a size of 0 tells the implementation not to waste that storage.

Before we end this section, there is a GNAT compiler switch you should know about. The `-gnatR?` switch instructs the compiler to list the representation details for the types, objects and subprograms in the compiled file(s). Both implementation-defined and user-defined representation details are presented. The '?' is just a placeholder and can be one of the following characters:

`[0|1|2|3|4][e][j][m][s]`

Increasing numeric values provide increasing amounts of information. The default is '1' and usually will suffice. See the GNAT User's Guide for Native Platforms for the details of the switch in section 4.3.15 Debugging Control, available online here:

https://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#debugging-control

You'll have to scroll down some to find that specific switch but it is worth finding and remembering. When you cannot understand what the compiler is telling you about the representation of something, this switch is your best friend.

31.6 Unchecked Programming

Ada is designed to be a reliable language by default, based as it is on static strong typing and high-level semantics. Many of the pitfalls that a developer must keep in the back of their mind with other languages do not apply in Ada, and are typically impossible. That protection extends to low-level programming as well, e.g., the Separation Principle. Nevertheless, low-level programming occasionally does require mechanisms that allow us to go beyond the safety net provided by the type rules and high-level language constructs.

One such mechanism (unchecked conversion) provides a way to circumvent the type system, a system otherwise firmly enforced by the compiler on our behalf. Note that by "circumventing the type system" we do not include so-called "checked" conversions. These conversions have meaningful semantics, and are, therefore, allowed by the language using a specific syntax. This conversion syntax is known as "functional" syntax because it looks like a function call, except that the "function" name is a type name, and the parameter is the object or value being converted to that type. These conversions are said to be "checked" because only specific kinds of types are allowed, and the compiler checks that such conversions are indeed between these allowed types.

Instead, this section discusses "unchecked" programming, so-called because the compiler does not check for meaningful semantics. There are multiple mechanisms for unchecked programming in Ada: in addition to circumventing the type system, we can also deallocate a previously-allocated object, and can create an access value without the usual checks. In all cases the responsibility for correct meaning and behavior rests on the developer. Very few, if any, checks are done by the compiler. If we convert a value to another type that generally makes no sense, for example a task object converted to a record type, we are on our own. If we deallocate an allocated object more than once, it is our fault and Bad Things inevitably result.

Likened to "escape hatches," the facilities for unchecked programming are explicit in Ada. Their use is very clear in the source code, and is relatively heavy: each mechanism is provided by the language in the form of a generic library subprogram that must be specified in a context clause ("with-clause") at the top of the file, and then instantiated prior to use, like any generic. For an introduction to generic units in Ada, see that section in the introductory Ada course: [Introduction to Ada](#) (page 115)

You should understand that the explicitly unchecked facilities in Ada are no more unsafe than the implicitly unchecked facilities in other languages. There's no safety-oriented reason to "drop down" to C, for example, to do low-level programming. For that matter, the low-level programming facilities in Ada are at least as powerful as those in other languages, and probably more so.

We will explore unchecked storage deallocation in a separate book so let's focus on unchecked type conversions.

Unchecked type conversions are achieved by instantiating this language-defined generic library function, a "child" of the root package named "Ada":

```
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
  function Ada.Unchecked_Conversion (S : Source) return Target
    with Pure, Nonblocking, Convention => Intrinsic;
```

The function, once instantiated and eventually invoked, returns the caller's value passed to S (of type Source) as if it is a value of type Target. That value can then be used in any way consistent with the Target type.

The two generic parameters, Source and Target, are defined in a manner that makes them very permissive in terms of the types they will accept when instantiated. To understand how, you need to understand a little bit of Ada's terminology and design for generic

unit parameters. (If you are already familiar with generic formal types and how they are matched, feel free to skip this material.)

First, the terminology. The type parameters defined by a generic unit are known as "generic formal types," or "generic formals" for short. Types Source and Target are the generic formals in the unit above. When instantiating such a generic, clients must specify a type for each generic formal type. The types specified by the client are known as "generic actual types," or "generic actuals" for short. You can remember that by the fact that the actuals are the types "actually" given to the generic unit to work with when instantiated. (You may laugh, but that mnemonic works.)

Now we're ready to discuss the language design concept. The idea is that the syntax of a generic formal type indicates what kind of generic actual is required for a legal instantiation. This is known as the "Contract Model" because we can think of the formal parameters as expressing a contract between the generic unit's implementation and the client code that instantiates the generic. The contract is enforced by the compiler, in that it will reject any instantiation that attempts to specify some actual type that does not match the formal's requirements.

For example, if the generic computes some value for any floating point type, that floating-point type would be declared as a generic formal type, and would be defined so that only some floating-point type could be used for the corresponding actual type:

```
generic
  type Real is digits <>;
```

The formal parameter syntax reflects the syntax of a floating-point type declaration, except that the `<>` (the "box") indicates that the generic does not care how many digits are available. The generic actual will be some floating point type and it will specify the number of decimal digits.

If instead we try to match that formal with some actual that is anything other than a floating-point type the compiler will reject the instantiation. Therefore, within the generic body, the implementation code can be written with the assurance that the characteristics and capabilities required of a floating point type will be available. That's the Contract Model in full: the requirements are a matter of the generic unit's purpose and implementation, so the formal parameters reflect those requirements and the compiler ensures they will be met.

Some generic units, though, do not require specifically numeric actual types. These generics can use less specific syntax for their formal types, and as a result, more kinds of actual types are permitted in the instantiations. Remember the Contract Model and this will make sense. The contract between the generic and the clients is, in this case, more permissive: it does not require a numeric type in order to implement whatever it does.

For illustration, suppose we want a generic procedure that will exchange two values of some type. What operations does the generic unit require in the implementation in order to swap two values? There are two: assignment, as you might expect, but also the ability to declare objects of the type (the "temporary" used to hold one of the values during the swap steps). As long as the body can do that, any type will suffice, so the generic formals are written to be that permissive. What is the syntax that expresses that permissiveness, you ask? To answer that, first consider simple, non-generic private types from the user's point of view. For example:

```
package P is
  type Foo is private;
  procedure Do_Something (This : Foo);
private
  type Foo is ... -- whatever
end P;
```

There are two "views" associated with the package: one for the "visible" part of the package

spec (declaration), known as the "partial" view, and one for the "private" part of the package spec and the package body, known as the "full" view. The differences between the two views are a function of compile-time visibility.

The partial view is what clients (i.e., users) of the package have: the ability to do things that a type name provides, such as declarations of objects, as well as some basic operations such as assignment, some functions for equality and inequality, some conversions, and whatever subprograms work on the type (the procedure `Do_Something` above). Practically speaking, that's about all that the partial view provides. That's quite a lot, in fact, and corresponds to the classic definition of an "abstract data type."

The code within the package private part and package body has the full view. This code has compile-time visibility to the full definition for type `Foo`, so there are additional capabilities available to this code. For example, if the full definition for `Foo` is as an array type, indexing will be available with the private part and body. If `Foo` is fully defined as some numeric type, arithmetic operations will be possible within the package, and so on.

Therefore, the full view provides capabilities for type `Foo` that users of the type cannot access via the partial view. Only the implementation for type `Foo` and procedure `Do_Something` have the potential to access them.

Now, back to the generic formal parameter. If the generic unit doesn't care what the actual type is, and just needs to be able do assignment and object declaration, a "generic formal private type" expresses exactly that:

```
generic
  type Item is private;
procedure Exchange( Left, Right : in out Item );

procedure Exchange( Left, Right : in out Item ) is
  Old_Left : Item;
begin
  Old_Left := Left;
  Left := Right;
  Right := Old_Left;
end Exchange;
```

Inside generic procedure `Exchange`, the view of type `Item` is as if `Item` were some private type declared in a package, with only the partial view available. But the operations provided by a partial view are sufficient to implement the body of `Exchange`: only assignment and object declaration are required. Any additional capabilities that the generic actual type may have — array indexing, arithmetic operators, whatever — are immaterial because they are not required. That's the Contract Model: only the specified view's required capabilities are important. Anything else the type can also do is not relevant.

But consider limited types. Those types don't allow assignment, by definition. Therefore, an instantiation that specified a limited actual type for the generic formal type `Item` above would be rejected by the compiler. The contract specifies the ability to do assignment so a limited type would violate the contract.

Finally, as mentioned, our `Exchange` generic needs to declare the "temporary" object `Old_Left`. A partial view of a private type allows that. But not all types are sufficient, by their name alone, to declare objects. Unconstrained array types, such as type `String`, are a familiar example: they require the bounds to be specified when declaring objects; the name `String` alone is insufficient. Therefore, such types would also violate the contract and, therefore, would be rejected by the compiler when attempting to instate generic procedure `Exchange`.

Suppose, however, that we have some other generic unit whose implementation does not need to declare objects of the formal type. In that case, a generic actual type that did not support object declaration (by the name alone) would be acceptable for an instantiation. The generic formal syntax for expressing that contract uses these tokens: (`<>`) in addition to the other syntax mentioned earlier:

```
generic
  type Foo(<>) is private;
```

In the above, the generic formal type `Foo` expresses the fact that it can allow unconstrained types — known as "indefinite types" — when instantiated because it will not attempt to use that type name to declare objects. Of course, the compiler will also allow constrained types (e.g., `Integer`, `Boolean`, etc.) in instantiations because it doesn't matter one way or the other inside the generic implementation. The Contract Model says that additional capabilities, declaring objects in this case, are allowed but not required. (There is a way to declare objects of indefinite types, but not using the type name alone. The unchecked facilities don't need to declare objects so we will not show how to do it.)

Now that you understand the Contract Model (perhaps more than you cared), we are ready to examine the generic formal type parameters for `Ada.Unchecked_Conversion`. Here's the declaration again:

```
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
  function Ada.Unchecked_Conversion (S : Source) return Target
    with Pure, Nonblocking, Convention => Intrinsic;
```

The two generic formal types, `Source`, and `Target`, are the types used for the incoming value and the returned value, respectively. Both formals are "indefinite, limited private types" in the jargon, but now you know what that means. Inside the implementation of the generic function, neither `Source` nor `Target` will be used to declare objects (the `(<>)` syntax). Likewise, neither type will be used in an assignment statement (the "limited" reserved word). And finally, no particular kind of type is required for `Source` or `Target` (the `private` reserved word). That's a fairly restricted usage within the generic implementation, but as a result the contract can be very permissive: the generic can be instantiated with almost any type. It doesn't matter if the actual is limited or not, private or not, and indefinite or not. The generic implementation doesn't need those capabilities to implement a conversion so they are not part of the contract expressed by the generic formal types.

What sort of type would be disallowed? Abstract types, and incomplete types. However, it is impossible to declare objects of those types, for good reasons, so unchecked conversion is never needed for them.

Note that the result value is returned by-reference whenever possible, in which case it is just a view of the `Source` bits in the formal parameter `S` and not a copy. For a `Source` type that is not a by-copy type, the result of an unchecked conversion will typically be returned by-reference (so that the result and the parameter `S` share the same storage); for a by-copy `Source` type, a copy is made.

The compiler can restrict instantiations but implementers are advised by the language standard to avoid them unless they are required by the target environment. For example, an instantiation for types for which unchecked conversion can't possibly make sense might be disallowed.

Clients can apply language- and vendor-defined restrictions as well, via `pragma Restrictions`. In particular, the language defines the `No_Dependence` restriction, meaning that no client's context clause can specify the unit specified. As a result no client can instantiate the generic for unchecked conversion:

```
pragma Restrictions (No_Dependence => Ada.Unchecked_Conversion);
```

hence there would be no use of unchecked conversion.

From the Contract Model's point of view most any type can be converted to some other type via this generic function. But practically speaking, some limitations are necessary. The following must all be true for the conversion effect to be defined by the language:

- `S'Size = Target'Size`
- `S'Alignment` is a multiple of `Target'Alignment`, or `Target'Alignment` is 0 (meaning no alignment required whatsoever)
- Target is not an unconstrained composite type
- S and Target both have a contiguous representation
- The representation of S is a representation of an object of the target subtype

We will examine these requirements in turn, but realize that they are not a matter of legality. Compilers can allow instantiations that violate these requirements. Rather, they are requirements for conversions to have the defined effect.

The first requirement is that the size (in bits) for the parameter S, of type Source, is the same as the size of the Target type. That's reasonable if you consider it. What would it mean to convert, for example, a 32-bit value to an 8-bit value? Which 8 bits should be used?

As a result, one of the few reasons for setting the size of a type (as opposed to the size of an object) is for the sake of well-defined unchecked conversions. We might make the size larger than it would need to be because we want to convert a value of that type to what would otherwise be a larger Target type.

Because converting between types that are not the same size is so open to interpretation, most compilers will issue a warning when the sizes are not the same. Some will even reject the instantiation. GNAT will issue a warning for these cases when the warnings are enabled, but will allow the instantiation. We're supposed to know what we are doing, after all. The warning is enabled via the specific `-gnatwz` switch or the more general `-gnatwa` switch. GNAT tries to be permissive. For example, 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. See the GNAT RM for the other details.

The next requirement concerns alignment. As we mentioned earlier, modern architectures tend to have strict alignment requirements. We can meaningfully convert to a type with a stricter alignment, or to a type with no alignment requirement, but converting in the other direction would require a copy.

Next, recall that objects of unconstrained types, such as unconstrained array types or discriminated record types, must have their constraints specified when the objects are declared. We cannot just declare a `String` object, for example, we must also specify the lower and upper bounds. Those bounds are stored in memory, logically as part of the `String` object, since each object could have different bounds (that's the point, after all). What, then, would it mean to convert some value of a type that has no bounds to a type that requires bounds? The third requirement says that it is not meaningful to do so.

The next requirement is that the argument for S, and the conversion target type Target, have a contiguous representation in memory. In other words, each storage unit must be immediately adjacent, physically, to the next logical storage unit in the value. Such a representation for any given type is not required by the language, although on typical modern architectures it is common. (The type `System.Storage_Elements.Storage_Array` is an exception, in that a contiguous representation is guaranteed.) An instance of `Ada.Unchecked_Conversion` just takes the bits of S and treats them as if they are bits for a value of type Target (more or less), and does not handle issues of segmentation.

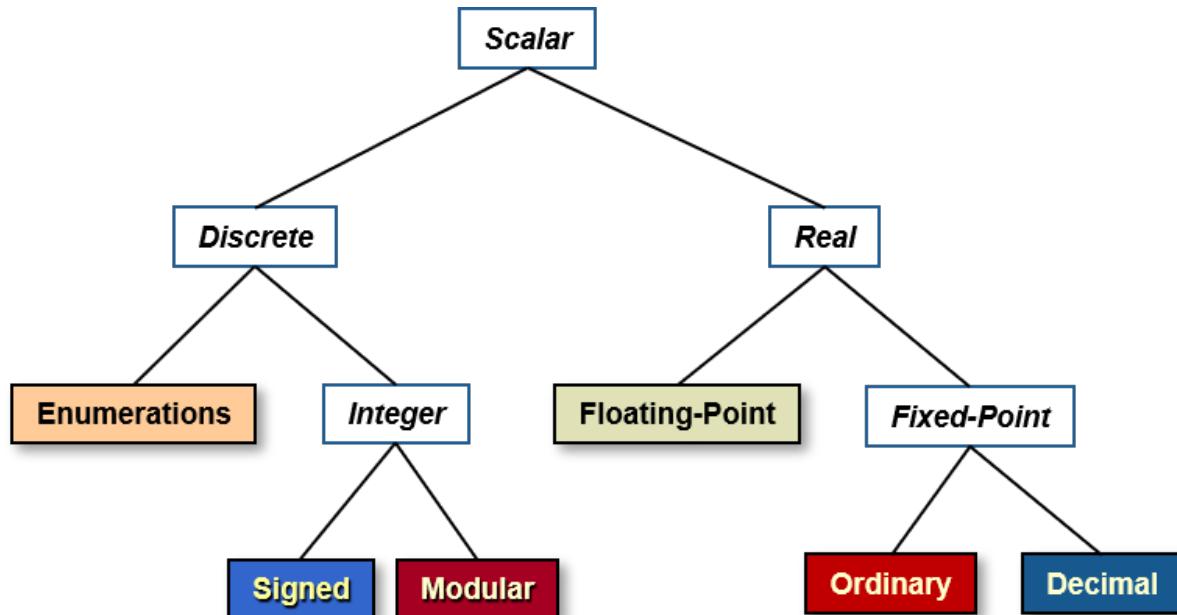
The last requirement merely states that the bits of the argument S, when treated as a value of type Target, must actually be a bit-pattern representing a value of type Target (strictly, the subtype). For example, with signed integers, any bit pattern (of the right size) represents a valid value for those types. In contrast, consider an enumeration type. By default, the underlying representational values are the same as the position values, i.e., starting at zero and increasing by one. But users can override that representation: they can start with any value and, although the values must increase, they need not increase by one:

```
type Toggle_Switch is (Off, On);
for Toggle_Switch use (Off => 0, On => 4);
```

If we convert an unsigned integer (of the right size) to a `Toggle_Switch` value, what would it mean if the Source value was neither 0 nor 4?

We've said that the instantiations are likely allowed, hence callable functions are created. If the above requirements are not met, what happens?

What happens depends on the Target type, that is, the result type for the conversion. Specifically, it depends on whether the target type is a "scalar" type. As we mentioned earlier, a scalar type is either a "discrete" type or a "real" type, which are themselves further defined, as the figure below indicates. Any other type is a non-scalar type, e.g., record types, access types, task types, and so on.



When the requirements for meaningful instantiations are not respected and the Target type is a scalar type, the result returned from the call is implementation defined and is potentially an invalid representation. For example, type `Toggle_Switch` is an enumeration type, hence it is a scalar type. Therefore, if we convert an unsigned integer (of the right size) to a `Toggle_Switch` value, and the Source value is neither 0 nor 4, the resulting value is an invalid representation. That's the same as an object of type `Toggle_Switch` that is never assigned a value. The random junk in the bits may or may not be a valid `Toggle_Switch` value. That's not a good situation, clearly, but it is well-defined: if it is detected, either `Constraint_Error` or `Program_Error` is raised. If the situation is not detected, execution continues using the invalid representation. In that case it may or may not be detected, near the call or later. For example:

```
with Ada.Unchecked_Conversion;
with Ada.Text_IO;  use Ada.Text_IO;
with Interfaces;   use Interfaces;

procedure Demo is

  type Toggle_Switch is (Off, On) with Size => 8;
  for Toggle_Switch use (Off => 1, On => 4);

  function As_Toggle_Switch is new Ada.Unchecked_Conversion
    (Source => Unsigned_8, Target => Toggle_Switch);
```

(continues on next page)

(continued from previous page)

```

T1 : Toggle_Switch;
T2 : Toggle_Switch;
begin
  T1 := As_Toggle_Switch (12); -- neither 1 nor 4
  if T1 = Off then
    Put_Line ("T1's off");
  else
    Put_Line ("T1's on");
  end if;
  T2 := T1;
  if T2 = Off then
    Put_Line ("T2's off");
  else
    Put_Line ("T2's on");
  end if;
  Put_Line (T2'Image);
end Demo;

```

In the execution of the code above, the invalid representation value in T1 is not detected, except that it is copied into T2, where it is eventually detected when 'Image is applied to T2. The invalid representation is not detected in the assignment statement or the comparison because we want the optimizer to be able to avoid emitting a check prior to every use of the value. Otherwise the generated code would be too slow. (The language explicitly allows this optimization.)

The evaluation of an object having an invalid representation value due to unchecked conversion is a so-called "bounded error" because the results at run-time are predictable and limited to one of those three possibilities: the two possible exceptions, or continued execution.

Continued execution might even work as hoped, but such code is not portable and should be avoided. A new vendor's compiler, or even a new version of a given vendor's compiler, might detect the situation and raise an exception. That happens, and it ends up costing developer time to make the required application code changes.

The possibilities get much worse when the result type is not a scalar type. In this case, the effect of the call — not the value returned by the call — is implementation defined. As a result, the possible run-time behavior is unpredictable and, consequently, from the language rules point of view anything is possible. Such execution is said to be "erroneous."

Why the difference based on scalar versus non-scalar types? Scalar types have a simple representation: their bits directly represent their values. Non-scalar types don't always have a simple representation that can be verified by examining their bits.

For example, we can have record types with discriminants that control the size of the corresponding objects because the record type contains an array component that uses the discriminant to set the upper bound. These record types might have multiple discriminants, and multiple dependent components. As a result, an implementation could have hidden, internal record components. These internal components might be used to store the starting address of the dependent components, for example, or might use pointers to provide a level of indirection. If an unchecked conversion did not provide correct values for these internal components, the effect of referencing the record object would be unpredictable.

Even a comparatively simple record type with one such dependent component is sufficient to illustrate the problem. There are no internal, hidden components involved:

```

with Ada.Unchecked_Conversion;
with Ada.Text_IO;           use Ada.Text_IO;
with System;                use System;  -- for Storage_Unit
with System.Storage_Elements; use System.Storage_Elements;

```

(continues on next page)

(continued from previous page)

```

procedure Demo_Errorous is

    subtype Buffer_Size is Storage_Offset range 1 .. Storage_Offset'Last;

    type Bounded_Buffer (Capacity : Buffer_Size) is record
        Content : Storage_Array (1 .. Capacity);
        Length   : Storage_Offset := 0;
    end record;

    procedure Show_Capacity (This : Bounded_Buffer);

    subtype OneK_Bounded_Buffer is Bounded_Buffer (Capacity => 1 * 1024);

    function As_OneK_Bounded_Buffer is new Ada.Unchecked_Conversion
        (Source => Storage_Array, Target => OneK_Bounded_Buffer);

    Buffer   : OneK_Bounded_Buffer;
    Sequence : Storage_Array (1 .. Buffer'Size / Storage_Unit);

    procedure Show_Capacity (This : Bounded_Buffer) is
    begin
        Put_line ("This.Capacity is" & This.Capacity'Image);
    end Show_Capacity;

begin
    Buffer := As_OneK_Bounded_Buffer (Sequence);
    Put_Line ("Buffer capacity is" & Buffer.Capacity'Image);
    Show_Capacity (Buffer);
    Put_Line ("Done");
end Demo_Errorous;

```

In the above, the type `Bounded_Buffer` has an array component `Content` that depends on the discriminant `Capacity` for the number of array components. This is an extremely common idiom. However, unchecked conversion is only meaningful, as defined earlier, when converting to constrained target types. `Bounded_Buffer` is not constrained, so we define a constrained subtype (`OneK_Bounded_Buffer`) for the sake of the conversion.

The specific `Buffer` object is 8320 bits ($1024 * 8$, plus $2 * 64$), as is the `Sequence` object, so the sizes are the same.

The alignment of `OneK_Bounded_Buffer` is 8, and `Storage_Array`'s alignment is 1, so the Target type is a multiple of the Source type, as required.

Both types have a contiguous representation, and the sequence of bytes can be a valid representation for the record type, although it certainly might not be valid. For example, if we change the discriminant from what the subtype specifies, we would have an invalid representation for that subtype.

So we can reasonably invoke an unchecked conversion between the array of bytes and the record type. However, as you can see in the code and as the compiler warns, we never assigned a value to the `Sequence` array object. The unchecked conversion from that `Sequence` of bytes includes the discriminant value, so it is very possible that we will get a discriminant value that is not 1K.

We can test that possibility by running the program. In the first call to `Put_Line`, the program prints the `Capacity` discriminant for the `Buffer` object. The compiler knew it was 1024, so it doesn't get the discriminant component from memory, it just directly prints 1024. However, we can force the compiler to query the discriminant in memory. We can pass `Buffer` to procedure `Show_Capacity`, which takes any `Bounded_Buffer`, and there query (print) the `Capacity` component under that different view. That works because the view inside the procedure `Show_Capacity` is as of `Bounded_Buffer`, in which the discriminant value is unknown at compile-time.

In the above examples, we are responsible for ensuring that the enumeration representation encoding and the record discriminant value are correct when converted from some other type. That's not too hard to recognize because we can literally see in the source code that there is something to be maintained by the conversions. However, there might be hidden implementation artifacts that we cannot see in the source code but that must be maintained nevertheless.

For example, the compiler's implementation for some record type might use dynamic memory allocations instead of directly representing some components. That would not appear in the source code. As a simpler example of invisible implementation issues, consider again our earlier record type:

```
type My_Int is range 1..10;
subtype S is Integer range 1..10;
type R is record
  M : My_Int;
  X : S;
  B : Boolean;
  C : Character;
end record;
```

If compiler allocates
in declaration order

Sample layout for a given compiler

3	2	1	0
			M
X	X	X	X
		C	B

R'Size will be 80 bits (10 bytes)
but all 12 are allocated to objects

As we discussed earlier, between the bytes that are allocated to the record components are some other bytes that are not used at all. As usual, the compiler must implement the language-defined equality operator for the record type. One way to implement that function would be to generate code that checks the equality for each component individually, ignoring any unused bytes. But suppose you have a large record type with many components. The code for checking record level equality will be extensive and inefficient. An alternative implementation for the compiler would be to use a "block compare" machine instruction to check the equality of the entire record at once, rather than component-by-component. That will be considerably more efficient because the block-compare instruction just compares the bits from one starting address to another ending address. But in that case the "unused" bytes are not skipped so the values within those bytes become significant. Comparison of those unused bytes will only work if their values are defined and assigned in each record object. Compilers that may use a block-comparison approach will, therefore, always set those unused bytes to a known value (typically zero). That is part of the valid representation for values of the type, and consequently must be maintained by our unchecked conversions. This being a non-scalar target type, failure to do so results in erroneous execution, i.e., undefined behavior. "There be dragons" as ancient maps of the unknown world once said.

As you can see, you should use unchecked conversions with considerable care and thought. Moreover, because unchecked programming is such a low-level activity, and has vendor-defined implementation issues, it is not only less portable than high-level coding, it is also less portable than other low-level programming. You will be well served if you limit the use of unchecked conversions overall. If your application code is performing unchecked conversions all over the code, something is very likely wrong, or at least very questionable. A well-designed Ada program should not need ubiquitous unchecked conversions.

That said, of course sometimes unchecked conversions are reasonable. But even then, it is better to isolate and hide their use via compile-time visibility controls. For example, instead of having clients invoke unchecked conversion instances many times, have a procedure that is invoked many times, and let the procedure body do the conversion. That way, the clients see a high-level specification of functionality, and, if the conversion needs to be changed later, there is only that one conversion usage (the procedure body) to change. This approach is really just another example of isolating and hiding code that might need to change in the future.

31.7 Data Validity

Our earlier demo program assigned an incorrect value via unchecked conversion into an object of an enumeration type that had non-standard representation values. The value assigned was not one of those representation values so the object had an invalid representation. Certain uses of an invalid representation value will be erroneous, and we saw that the effect of erroneous execution was unpredictable and unbounded.

That example was somewhat artificial, for the sake of illustration. But we might get an invalid value in a real-world application. For example, we could get an invalid value from a sensor. Hardware sensors are frequently unreliable and noisy. We might get an invalid value from a call to an imported function implemented in some other language. Whenever an assignment is aborted, the target of the assignment might not be fully assigned, leading to so-called "abnormal" values. Other causes are also possible. The problem is not unusual in low-level programming.

How do we avoid the resulting bounded errors and erroneous execution?

In addition to assignment statements, we can safely apply the `Valid` attribute to the object. This language-defined attribute returns a Boolean value indicating whether or not the object's value is a valid representation for the object's subtype. (More details in a moment.) There is no portable alternative to check an object's validity. Here's an example:

```
with Ada.Unchecked_Conversion;
with Ada.Text_Io;  use Ada.Text_Io;
with Interfaces;   use Interfaces;
with System;

procedure Demo_Validity_Check is

  type Toggle_Switch is (Off, On) with Size => 8;
  for Toggle_Switch use (Off => 1, On => 4);

  T1 : Toggle_Switch;

  function Sensor_Reading (Default : Toggle_Switch) return Toggle_Switch is

    function As_Toggle_Switch is new Ada.Unchecked_Conversion
      (Source => Unsigned_8, Target => Toggle_Switch);

    Result : Toggle_Switch;
    Sensor : Unsigned_8;
    -- for Sensor'Address use System'To_Address (...);

  begin
    Result := As_Toggle_Switch (Sensor);
    return (if Result'Valid then Result else Default);
  end Sensor_Reading;

begin
  T1 := Sensor_Reading (Default => Off); -- arbitrary
  Put_Line (T1'Image);
end Demo_Validity_Check;
```

In the above, `Sensor_Reading` is the high-level, functional API provided to clients. The function hides the use of the unchecked conversion, and also hides the memory-mapped hardware interface named `Sensor`. We've commented out the address clause since we don't really have a memory mapped device available. You can experiment with this program by changing the code to assign a value to `Sensor` (e.g., when it is declared). It is an unsigned 8-bit quantity so any value in the corresponding range would be allowed.

In addition to checking for a valid representation, thus preventing the bounded error, `Valid`

also checks that the object is not abnormal, so erroneous execution can be prevented too. (It also checks that any subtype predicate defined for the Target type is also satisfied, but that's a lesson for another day.)

However, the `Valid` attribute can be applied only to scalar objects. There is no language-defined attribute for checking objects of composite types. That's because it would be very hard to implement for some types, if not impossible. For example, given a typical run-time model, it is impossible to check the validity of an access value component. Therefore, you must individually check the validity of scalar record or array components.

At least, you would have to check them individually in standard Ada. GNAT defines another Boolean attribute, named `Valid_Scalars`, to check them all for us. This attribute returns `True` if the evaluation of `Valid` returns `True` for every scalar subcomponent of the enclosing composite type. It also returns `True` when there are no scalar subcomponents. See the GNAT RM for more information.

CHAPTER
THIRTYTWO

MULTI-LANGUAGE DEVELOPMENT

Software projects often involve more than one programming language. Typically that's because there is existing code that already does something we need done and, for that specific code, it doesn't make economic sense to redevelop it in some other language. Consider the rotor blade model in a high-fidelity helicopter simulation. Nobody touches the code for that model except for a few specialists, because the code is extraordinarily complex. (This complexity is unavoidable because a rotor blade's dynamic behavior is so complex. You can't even model it as one physical piece because the tip is traveling so much faster than the other end.) Complex and expensive models like that are a simulator company's crown jewels; their cost is meant to be amortized over as many projects as possible. Nobody would imagine redeveloping it simply because a new project is to be written in a different language.

Therefore, Ada includes extensive facilities to "import" foreign entities into Ada code, and to "export" Ada entities to code in foreign languages. The facilities are so useful that Ada has been used purely as "glue code" to allow code written in two other programming languages to be used together.

You've already seen an introduction to Ada and C code working together in the "["Interfacing" section of the Ada introductory course](#)" (page 165). If you have not seen that material, be sure to see it first. We will cover some further details not already discussed there, and then go into the details of the facilities not covered elsewhere, but we assume you're familiar with it.

The Ada foreign language interfacing facilities include both "general" and "language-specific" capabilities. The "general" facilities are known as such because they are not tied to any specific language. These pragmas and aspects work with any of the supported foreign languages. In contrast, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. For example, as you saw in that "Interfacing" section, there is a package with a number of declarations for C types, such as `int`, `float`, and `double`, as well as C "strings", with subprograms to convert back and forth between them and Ada's string type. Other languages are also supported, both by the Ada Standard and by vendor additions. You will frequently use both the "general" and the "language-specific" facilities together.

All these interfacing capabilities are defined in Annex B of the language standard. Note that Annex B is not a "Specialized Needs" annex, unlike some of the other annexes. The Specialized Needs annexes are wholly optional, whereas all Ada implementations must implement Annex B. However, some parts of Annex B are optional, so more precisely we should say that every implementation must support all the required features of Annex B. That comes down mainly to the package `Interfaces` (more on that package in a moment). However, if an implementation does implement any optional part of Annex B, it must be implemented as described by the standard, or with less functionality. An implementation cannot use the same name for some facility (aspect, etc.) but with different semantics. That's true of the Specialized Needs annexes too: not every part need be implemented, but any part that is implemented must conform to the standard. In practice, for Annex B, all implementations provide the required parts, but not all provide support for all the "language-specific" foreign

languages' interfaces. The vendors make a business decision for the optional parts, just as they do regarding the Specialized Needs annexes.

32.1 General Interfacing

In the "Interfacing" section of the Ada introductory course you saw that Ada defines aspects and pragmas for working with foreign languages. These aspects and pragmas are functionally interchangeable, and we will use whichever one of the two that is most convenient in our discussion. The pragmas are officially "obsolescent," but that merely means that a newer approach is available, in this case the corresponding aspects. You can use either one without concern for future support because language constructs that are obsolescent are not removed from the language. Any compiler that supports such constructs will almost certainly support them forever, for the sake of not invalidating existing customers' code. The pragmas have been in the language since Ada 95 so there's a lot of existing code using them. Changing the compiler isn't cost-free, after all, so why spend the money to potentially lose a customer? Likewise, a brand new compiler will also probably support them, for the sake of potentially gaining a customer.

The general interfacing facility consists of these aspects and pragmas, specifically Import, Export, and Convention. As you saw in the Ada Introduction course, Import brings a foreign entity into Ada code, Export does the opposite, and Convention supplies additional information and directives to the compiler. We will go into the details of each.

Regardless of whether the Ada code is importing or exporting some entity, there will be an Ada declaration for that entity. That declaration tells the compiler how the entity can be used, as usual. The interfacing aspects and pragmas are then applied to these Ada declarations.

If we are exporting, then the entity is implemented in Ada. For a subprogram that means there will also be a subprogram body matching the declaration, and the compiler will enforce that requirement as usual. In contrast, if we are importing a subprogram, then it is not implemented in Ada, and therefore there will be no corresponding subprogram body for the Ada declaration. The compiler would not allow it if we tried. In that case the Import is the subprogram's completion.

Subprograms often have a separate declaration. Sometimes that's required, for example when we want to include a subprogram as part of a package's API, but at other times it is optional. Remember that a subprogram body acts as a corresponding declaration when there is no separate declaration defined. Thus, either way, we have a subprogram declaration available for the interfacing aspects and/or pragmas.

For data that are imported or exported, we'll have the declaration of the object in Ada to which we can apply the necessary interfacing aspects/pragmas. But we will also have the types for these objects, and as you will see, the types can be part of interfacing too.

32.1.1 Aspect/Pragma Convention

As you saw in the "*Interfacing*" section of the Ada introductory course (page 165), when importing and exporting you'll also specify the "convention" for the entity in question. The pragmas for importing and exporting include a parameter for this purpose. When using the aspects, you'll specify the Convention aspect too.

For types, though, you will specify the Convention aspect/pragma alone, without Import or Export. In this case the convention specifies the layout for objects of that type, presumably a layout different than the Ada compiler would normally use. You would need to specify this other layout either because you're going to later declare and export an object of the type, or because you are going to declare an object of the type and pass it as an argument to an imported subprogram.

For example, Ada specifies that multi-dimensional arrays are represented in memory in row-major order. In contrast, the Fortran standard specifies column-major order. If we want to define a type in Ada that can be used for passing parameters to Fortran routines, we need to specify that convention for the type. For example:

```
type Matrix is array (Rows, Columns) of Float
with Convention => Fortran;
```

(Rows and Columns are user-defined discrete subtypes.)

As a result when we declare Matrix objects the Ada compiler will use the column-major layout. That makes it possible to pass objects of the type to imported Fortran subprograms because the formal parameter will also be of type Matrix. The imported Fortran routine will then see the parameter in memory as it expects to see it. So although you wouldn't need to import or export a type itself, you might very well import or export an object of the type, or pass it as a argument.

When Convention is applied to subprograms, a natural mistake is to think that we are specifying the programming language used to implement the subprogram. In reality, the convention indicates the subprogram calling convention, not the implementation language. The calling convention specifies how parameters are passed to and from subprogram calls, how result values for functions are returned, the order that parameters are pushed on the call stack, how dynamically-sized parameters are passed, and so on. Ordinarily these are matters you don't need to consider because you're working within a single convention automatically, in other words the one used by the Ada compiler you're using.

To illustrate that the convention is not the implementation language, consider a subprogram that we intend to import and call from Ada. This imported routine is implemented in assembly language, but, in addition, let's say it is written to use the same calling convention as the Ada compiler we are using for Ada code. Therefore, the calling convention would be Ada even though the implementation is in assembler.

```
procedure P (X : Integer) with
  ...
  Convention => Ada,
  ...
```

In the example above, Ada is known as a convention identifier, as is Fortran in the earlier example. Convention identifiers are defined by the Ada language standard, but also by Ada vendors.

The Ada standard defines two convention identifiers: Ada (the default), and Intrinsic. In addition, Annex B defines convention identifiers C, COBOL, and Fortran. Support for these Annex B conventions is optional.

GNAT supports the standard and Annex B conventions, as well as the following: Assembler, "C_PLUS_PLUS" (or CPP), Stdcall, WIN32, and a few others. C_PLUS_PLUS is the convention identifier required by the standard when C++ is supported. (Convention identifiers are actual identifiers, not strings, so they must obey the syntax rules for identifiers. "C++" would not be a valid identifier.) See the GNAT User Guide for those other GNAT-specific conventions.

Stdcall and WIN32 actually do specify a particular calling convention, but for those convention identifiers that are language names, how do we get from the name to a calling convention?

The ultimate requirement for any calling convention is compatibility with the Ada compiler we are using. Specifically, the Ada compiler must recognize what the calling convention specifies, and support importing and exporting subprograms with that convention applied.

For the Ada convention that's simple. There is no standard calling convention for Ada. Convention Ada simply means the calling convention applied by the Ada compiler we happen to be using. (We'll talk about Intrinsic shortly.)

So far, so good. But how do we get from those other language names to corresponding calling conventions? There is no standard calling convention for, say, C, any more than there is a standard calling convention for Ada.

In fact we don't get to the calling convention, at least not directly. What the language name in the convention identifier actually tells us is that, when that convention is supported, there is a compiler for that foreign language that uses a calling convention known to, and supported by, the Ada compiler we are using. The Ada compiler vendor defines which languages it supports, after all. For example, when supported, convention C means that there is a compatible C compiler known to the Ada compiler vendor. For GNAT you can guess which C compiler that might be.

It's actually pretty straightforward once you have the big picture. If the convention is supported, the Ada compiler in use knows of a compiler for that language with which it can work. Annex B just defines some convention identifiers for the sake of portability.

But suppose a given Ada compiler supports more than one vendor for a given programming language? In that case the Ada compiler would define and support multiple convention identifiers for the same programming language. Presumably these identifiers would be differentiated by the compiler vendors' names. Thus we might have available conventions `GNU_Fortran` and `Intel_Fortran` if both were supported. The `Fortran` convention identifier would then indicate the default vendor's compiler.

The `Intrinsic` calling convention represents subprograms that are "built in" to the compiler. When such a subprogram is called the compiler doesn't actually generate the code for an out-of-line call. Instead, the compiler emits the assembly code — often just a single instruction — corresponding to the intrinsic subprogram's name. There will be a separate declaration for the subprogram, but no actual subprogram body containing a sequence of statements. The compiler just knows what to emit in place of the call.

For example:

```
function Shift_Left
  (Value : Unsigned_16;
   Amount : Natural)
  return Unsigned_16
  with ..., Convention => Intrinsic;
```

The effect is much like a subprogram call that is always in-lined, except that there's no body for the subprogram. In this example the compiler simply issues a shift-left instruction in assembly language.

You'll see the `Intrinsic` convention applied to many language-defined subprograms. For example:

```
generic
  type Source(<>) is limited private;
  type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target
  with ..., Convention => Intrinsic;
```

Thus when we call an instantiation of `Ada.Unchecked_Conversion` there is no actual call made to some subprogram. The compiler just treats the bits of `S` as a value of type `Target`.

Intrinsic subprograms are a good way to access interesting capabilities of the target hardware, without having to write the assembly language yourself (although we will show how to do that, later, directly in Ada). For example, some targets provide an instruction that atomically compares and swaps a value in memory. Ada 2022 just added a standard package for this, but before that we could use the following to access a gcc built-in:

```
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
```

(continues on next page)

(continued from previous page)

```
-- Returns an indication of whether the swap took place.
```

```
function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean
with Convention => Intrinsic,
  ...
```

We would specify additional aspects beyond that of Convention but these have not yet been discussed. That's what the ellipses indicate in the various examples above.

32.1.2 Aspect/Pragma Import and Export

You've already seen these aspects in the Ada Introduction course, but for completeness: Import brings a foreign entity into Ada code, and Export makes an Ada entity available to foreign code. In practice, these entities consist of objects and subprograms, but the language doesn't impose many restrictions. It is up to the vendor to decide what makes sense for their specific target.

The aspects Import and Export are so-called Boolean aspects because their value is either **True** or **False**. For example:

```
Obj : Matrix with
  Export => True,
  ...
```

For any Boolean-valued aspect the default is **True** so you only need to give the value explicitly if that value is **False**. There would be no point in doing that in these two cases, of course. Hence we just give the aspect name:

```
Obj : Matrix with
  Export,
  ...
```

Recall that objects of some types are initialized automatically during the objects' elaboration, unless they are explicitly initialized as part of their declarations. Access types are like that, for example. Objects of these types are default initialized to **null** as part of ensuring that their values are always meaningful (absent unchecked conversion).

```
type Reference is access Integer;

Obj : Reference;
```

In the above the value of Obj is **null**, just as if we had explicitly set it that way.

But that initialization is a problem if we are importing an object of an access type. Presumably the value is set by the foreign code, so automatic initialization to null would overwrite the incoming value. Therefore, the language guarantees that implicit initialization won't be applied to imported objects.

```
type Reference is access Integer;

Obj : Reference with Import;
```

Now the value of Obj is whatever the foreign code sets it to, and is not, in other words, overwritten during elaboration of the declaration.

32.1.3 Aspect/Pragma External_Name and Link_Name

For an entity with a **True** Import or Export aspect, we can also specify a so-called external name or link name. These names are specified via aspects External_Name and Link_Name respectively.

An external name is a string value indicating the name for some entity as known by foreign language code. For an entity that Ada code imports, this is the name that the foreign code declares it to be. For an entity that Ada code exports, this is the name that the foreign code is told to use. This string value is exactly the name to be used, so if you misspell the name the link will fail. For example:

```
function Sync_Val_Compare_And_Swap_Bool_8
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean
with
  Import,
  Convention    => Intrinsic,
  External_Name  => "__sync_bool_compare_and_swap_1";
```

The External_Name and Link_Name values are strings because the foreign unit names don't necessarily follow the Ada rules for identifiers (the leading underscores in this case). Note that the ending digit in the name above is different from the declared Ada name.

Usually, the name of the imported or exported entity is precisely known and hence exactly specified by External_Name. Sometimes, however, a compilation system may have a linker "preprocessor" that augments the name actually used by the linkage step. For example, an implementation might always prepend "_" and then pass the result to the system linker. In that case we don't want to specify the exact name. Instead, we want to provide the "starting point" for the name modification. That's the purpose of the aspect Link_Name.

If you don't specify either External_Name or Link_Name the compilation system will choose one in some implementation-defined manner. Typically this would be the entity's defining name in the Ada declaration, or some simple transformation thereof. But usually we know the name exactly and so we use External_Name to give it.

As you can see, it really wouldn't make sense to specify both External_Name and Link_Name since the semantics of the two conflict. But if both are specified for some reason, the External_Name value is ignored.

Note that Link_Name cannot be specified for Intrinsic subprograms because there is no actual unit being linked into the executable, because intrinsics are built-in. In this case you must specify the External_Name.

Finally, because you will see a lot the pragma usage we should go into enough detail so that you know what you're looking at when you see them.

Pragma Import and pragma Export work almost like a subprogram call. Parameters cannot be omitted unless named notation is used. Reordering the parameters is not permitted, however, unlike subprogram calls.

The BNF syntax is as follows. We show Import, but Export has identical parameters:

```
pragma Import(
  [Convention =>] convention_identifier,
  [Entity =>] local_name
  [, [External_Name =>] external_name_string_expression]
  [, [Link_Name =>] link_name_string_expression]);
```

As you can see, the parameters correspond to the individual aspects Convention, External_Name, and Link_Name. When using aspects you don't need to say which Ada entity

you're applying the aspects to, because the aspects are part of the entity declaration syntax. In contrast, the pragma is distinct from the declaration so we must specify what's being imported or exported via the Entity parameter. That's the declared Ada name, in other words. Note that both the External_Name and Link_Name parameters are optional.

Here's that same built-in function, using the pragma to import it:

```
-- Perform an atomic compare and swap: if the current value of
-- Destination.all is Comparand, then write New_Value into Destination.all.
-- Returns an indication of whether the swap took place.

function Sync_Val_Compare_And_Swap_Boolean
  (Destination : access Unsigned_8;
   Comparand   : Unsigned_8;
   New_Value   : Unsigned_8)
  return Boolean;

pragma Import (Intrinsic,
               Sync_Val_Compare_And_Swap_Boolean,
               "__sync_bool_compare_and_swap_1");
```

The first pragma parameter is for the convention. The next parameter, the Entity, is the Ada unit's declared name. The last parameter is the external name. The compiler either knows what we are referencing by that external name or it will reject the pragma. As we mentioned before, the string value for the name is not required to match the Ada unit name.

You will see later that there are other convention identifiers as well, but we will wait for the *Specific Interfacing section* (page 443) to introduce those.

32.1.4 Package Interfaces

Package Interfaces must be provided by all Ada implementations. The package is intended to provide types that reflect the actual numeric types provided by the target hardware. Of course, the standard has no way to know what hardware is involved, therefore the actual content is implementation-defined. But even so, it is possible to standardize the names for these types, and that is what the language standard does.

Specifically, the standard defines the format for the names for the hardware's signed and modular (unsigned) integer types, and for the floating-point types.

The signed integers have names of the form Integer_n, where n is the number of bits used by the machine-supported type. The type for an eight-bit signed integer would be named Integer_8, for example, and then Integer_16 and so on for the larger types, for as many as the target machine supports.

Likewise, for the unsigned integers, the names are of the form Unsigned_n, with the same meaning for n. The colloquial eight-bit "byte" would be named Unsigned_8, with Unsigned_16 for the 16-bit version, and so on, again for as many as the machine supports.

For floating-point types it is harder to talk about a format that is sufficiently common to standardize. The IEEE floating-point standard is well known and widely used, however, so if the machine does support the IEEE format that name can be used. Such types would be named IEEE_Float_n, again with the same meaning for n. Thus we might see declarations for types IEEE_Float_32 and IEEE_Float_64 and so on, for all the machine supported floating-point types.

In addition to these type declarations, for the unsigned integers only, there will be declarations for shift and rotate operations provided as intrinsic functions.

The resulting package declaration might look something like this:

```

package Interfaces is

    type Integer_8  is range -2 ** 7 .. 2 ** 7 - 1;
    type Integer_16 is range -2 ** 15 .. 2 ** 15 - 1;
    type Integer_32 is range -2 ** 31 .. 2 ** 31 - 1;

    ...

    type Unsigned_8 is mod 2 ** 8;

    function Shift_Left  (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
    function Shift_Right (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
    function Rotate_Left (Value : Unsigned_8;  Amount : Natural) return Unsigned_8;
    function Rotate_Right (Value : Unsigned_8; Amount : Natural) return Unsigned_8;
    function Shift_Right_Arithmetic (Value : Unsigned_8; Amount : Natural)
        return Unsigned_8;

    type Unsigned_16 is mod 2 ** 16;

    function Shift_Left  (Value : Unsigned_16;  Amount : Natural)
        return Unsigned_16;
    function Shift_Right (Value : Unsigned_16;  Amount : Natural)
        return Unsigned_16;
    ...

    type Unsigned_32 is mod 2 ** 32;

    function Shift_Left  (Value : Unsigned_32;  Amount : Natural)
        return Unsigned_32;
    function Shift_Right (Value : Unsigned_32;  Amount : Natural)
        return Unsigned_32;
    ...

    type IEEE_Float_32 is digits 6;
    type IEEE_Float_64 is digits 15;
    ...

end Interfaces;

```

As you can see, when you need to write code in terms of the hardware's numeric types, this package is a great resource. There's no need to declare your own UInt32 type, for example, although of course you could, trivially:

```
type UInt32 is mod 2 ** 32;
```

But if you do, realize that you won't get the shift and rotate operations for your type. Those are only defined for the types in package Interfaces. If you do need to declare such a type, and you do want the additional shift/rotate operations, use inheritance:

```
type UInt32 is new Interfaces.Unsigned_32;
```

GNAT also defines a pragma, as an alternative to inheritance:

```
type UInt32 is mod 2 ** 32;
pragma Provide_Shift_Operators (UInt32);
```

The approach using inheritance is preferable because it is portable, all other things being equal.

One reason to make up your own unsigned type is that you need one that does not in fact

reflect the target hardware's numeric types. For example, a hardware device register might have gaps of bits that are currently not used by the device. Those gaps are frequently not the size of a type declared in package Interfaces. We might need an Unsigned_3 type, for example. That's a reasonable thing to do.

32.2 Language-Specific Interfacing

In addition to the aspects and pragmas for importing and exporting entities that work with any language, Ada also defines standard language-specific facilities for interfacing with a set of foreign languages. The standard defines which languages, but vendors can (and do) expand the set.

Specifically, the "language-specific" interfacing facilities are collections of Ada declarations that provide Ada analogues for specific foreign language types and subprograms. Package Interfaces is the root package for a hierarchy of packages that organize these declarations by language, with one or more child packages per language.

Note that the declarations within package Interfaces are, by definition, compile-time visible to any child package in the subsystem. Thus whenever one of the language-specific packages needs to mention the machine types they are automatically available.

The standard defines specific support for foreign languages C, COBOL, and Fortran. Thus there are one or more child packages rooted at Interfaces that have those language names as their child package names: Interfaces.C, Interfaces.COBOL, and Interfaces.Fortran.

The material below will focus on C and, to a lesser extent, Fortran, ignoring altogether the support for COBOL. That's not because COBOL is unimportant. There is a lot of COBOL business software out there in use. Rather, we skip COBOL because it is not relevant to embedded systems. Similarly, although Fortran is extensively used, especially in high-performance computing, it is not used extensively in embedded systems. We will provide some information about the Fortran support but will not dwell on it.

Even though we do not consider C to be appropriate for large development projects, neither technically nor economically, it has its place in small, low-criticality embedded systems. Ada developers can profit from existing device drivers and mature libraries coded in C, for example. Hence interfacing to it is important.

What about C++? Interfacing to C++ is tricky compared to C, because of the vendor-defined name-mangling, automatic invocations of constructors and destructors, exceptions, and so on. Generally, interfacing with C++ code can be facilitated by preventing much of those difficulties using the `extern "C" { ... }` linkage-specification. Doing so then makes the bracketed C++ code look like C, so the C interfacing facilities then can be used.

32.2.1 Package Interfaces.C

The child package Interfaces.C supports interfacing with units written in the C programming language. Support is in the form of Ada constants and types, and some subprograms. The constants correspond to C's limits.h header file, and the Ada types correspond to types for C's int, short, unsigned_short, unsigned_long, unsigned_char, size_t, and so on. There is also support for converting Ada's type `String` to/from `char_array`, and similarly for type `Wide_String`, etc.

It's a large package so we will elide parts. The idea is to give you a feel for what's there. If you want the details, see either the Ada reference manual or bring up the source code in GNAT Studio.

```

package Interfaces.C is

    -- Declaration's based on C's <limits.h>

    CHAR_BIT : constant := 8;
    SCHAR_MIN : constant := -128;
    SCHAR_MAX : constant := 127;
    UCHAR_MAX : constant := 255;

    -- Signed and Unsigned Integers. Note that in GNAT, we have ensured that
    -- the standard predefined Ada types correspond to the standard C types

    type int    is new Integer;
    type short is new Short_Integer;
    type long   is range -(2 ** (System.Parameters.long_bits - Integer'(1)))
        .. +(2 ** (System.Parameters.long_bits - Integer'(1))) - 1;
    type long_long is new Long_Long_Integer;

    type signed_char is range SCHAR_MIN .. SCHAR_MAX;
    for signed_char'Size use CHAR_BIT;

    type unsigned           is mod 2 ** int'Size;
    type unsigned_short     is mod 2 ** short'Size;
    type unsigned_long      is mod 2 ** long'Size;
    type unsigned_long_long is mod 2 ** long_long'Size;

    ...

    -- Floating-Point

    type C_float      is new Float;
    type double       is new Standard.Long_Float;
    type long_double is new Standard.Long_Long_Float;

    -----
    -- Characters and Strings --
    -----


    type char is new Character;

    nul : constant char := char'First;

    function To_C (Item : Character) return char;
    function To_Ada (Item : char)      return Character;

    type char_array is array (size_t range <>) of aliased char;
    for char_array'Component_Size use CHAR_BIT;

    ...

end Interfaces.C;

```

The primary purpose of these types is for use in the formal parameters of Ada subprograms imported from C or exported to C. The various conversion functions can be called from within Ada to manipulate the actual parameters.

When writing the Ada subprogram declaration corresponding to a C function, an Ada procedure directly corresponds to a void function. An Ada procedure also corresponds to a C function if the return value is always to be ignored. Otherwise, the Ada declaration should be a function.

As we said, the types declared in this package can be used as the formal parameter types. That is the intended and recommended approach. However, some Ada types naturally

correspond to C types, and you might see them used instead of those from `Interfaces.C`. Type `int` is the C native integer type for the target, for example, as is type `Integer` in Ada. Likewise, C's type `float` and type Ada's `Float` are likely compatible. GNAT goes to some lengths to maintain compatibility with C, since the two gcc compilers share so much internal technology. Other vendors might not do so. Best practice is use the types in `Interfaces.C` for your parameters.

Of course, the types in `Interfaces.C` are not sufficient for all uses. You will often need to use user-defined types for the formal parameters, such as enumeration types and record types.

Ada enumeration types are compatible with C's enums but note that C requires enum values to be the size of an `int`, whereas Ada does not. The Ada compiler uses whatever sized machine type will support the specified number of enumeral values. It might therefore be smaller than an `int` but it might also be larger. (Declaring more enumeration values than would fit in an `integer` is unlikely except in tool-generated code, but it is possible.) For example:

```
type Small_Enum is (A, B, C);
```

If we printed the object size for `Small_Enum` we'd get 8 (on a typical machine with GNAT). Therefore, applying the aspect `Convention` to the Ada enumeration type declaration is a good idea:

```
type Small_Enum is (A, B, C) with Convention => C;
```

Now the object size will be 32, the same as `int`.

Speaking of enumeration types, note that Ada 2022 added a boolean type to `Interfaces.C` named `C_Bool` to match that of C99, so you should use it instead of Ada's `Boolean` type for formal parameters.

A simple Ada record type is compatible with a C struct, but remember that the Ada compiler is allowed to reorder the record components. The compiler would do that if it saw that the layout was inefficient, but the point here is that the compiler could do it silently. As a result, you should specify the record layout explicitly using a record representation clause, matching the layout of the C struct in question. Then there will be no question of the layouts matching. Once your record types get more complicated, for example with discriminants or tagged record extensions, things get tricky. Your best bet is to stick with the simple cases when interfacing to C.

Some types that you might think would correspond do not, at least not necessarily. For example, an Ada access type's value might be represented as a simple address, but it might not. In GNAT, an access value designating a value of some unconstrained array type (e.g., `String`) is comprised of two addresses, by default. One designates the characters and the other designates the bounds. You can override that with a pragma, but you must know to do so. For example, if we run the following program, we will see that the object size for the access type `Name` is twice the object size of `System.Address`:

```
with Ada.Text_IO;  use Ada.Text_IO;
with System;       use System;

procedure Demo is

    type Name is access String;

begin
    Put_Line (Address'Object_Size'Image);
    Put_Line (Name'Object_Size'Image);
end Demo;
```

Some Ada types simply have no corresponding type in C, such as record extensions, task

types, and protected types. You'll have to pass those as an "opaque" type, usually as an address. It isn't clear that a C function would know what to do with values of these types, but the general notion of passing an opaque type as an address is useful and not uncommon. Of course, that approach forgoes all type safety, so avoid it when possible.

In addition to the types for the formal parameters, you'll also need to know how parameters are passed to and from C functions. That affects the parameter profiles on both sides, Ada and C. The text in Annex B for Interfaces.C specifies how parameters are to be passed back and forth between Ada and C so that your subprogram declarations can be portable. That's the approach for each supported programming language, i.e., in the discussion of the corresponding child package under Interfaces.

The rules are expressed in terms of scalar types, "elementary" types, array types, and record types. Remember that scalar types are composed of the discrete types and the real types, so we're talking about the signed and modular integers, enumerations, floating-point, and the two kinds of fixed-point types. The "elementary" types consist of the scalars and access types. The rules are fairly intuitive, but throw in Ada's access parameters and parameter modes and some subtleties arise. We won't cover all the various rules but will explore some of the subtleties.

First, the easy cases: mode **in** scalar parameters, such as `int`, as simply passed by copy. Scalar parameters are passed by copy anyway in Ada so the mechanism aligns with C in a straightforward manner. A record type `T` is passed by reference, so on the C side we'd see `t*` where `t` is a C struct corresponding to `T`. A constrained array type in Ada with a component type `T` would correspond to a C formal parameter `t*` where `t` corresponds to `T`. An Ada access parameter **access** `T` corresponds on the C side to `t*` where `t` corresponds to `T`. And finally, a private type is passed according to the full definition of the type; the fact that it is private is just a matter of controlling the client view, being private doesn't affect how it is passed. There are other simple cases, such as access-to-subprogram types, but we can leave that to the Annex.

Now to the more complicated cases. First, some C ABIs (application binary interfaces) pass small structs by copy instead of by reference. That can make sense, in particular when the struct is small, say the size of an address or smaller. In that case there's no performance benefit to be had by passing a reference. When that situation applies, there is another convention we have not yet mentioned: `C_Pass_By_Copy`. As a result the record parameter will be passed by copy instead of the default, by reference (i.e., `T` rather than `*T`), as long as the mode is **in**. For example:

```
type R2 is record
    V : int;
end record;
with Convention => C_Pass_By_Copy;

procedure F2 (P : R2) with
    Import,
    Convention => C,
    External_Name => "f2";
```

```
struct R2 {
    int V;
};

void f2 (R2 p);
```

On the C side we expect that `p` is passed by copy and indeed that is how we find it. That said, passing record values to structs by reference is the more common programmer choice. Like arrays, records are typically larger than an address. The point here is that the Ada code can be configured easily to match the C code.

Next, consider passing array values, both to and from C. When passing an array value to C, remember that Ada array types have bounds. Those bounds are either specified at compile

time when they are declared, or, for unconstrained array types, specified elsewhere, at run-time.

Array types are not first-class types in C, and C has no notion of unconstrained array types, or even of upper bounds. Therefore, passing an unconstrained array type value is interesting. One approach is to avoid them. Instead, declare a sufficiently large constrained array as a subtype of the unconstrained array type, and then just pass the actual upper bound you want, along with the array object itself.

```
type List is array (Integer range <>) of Interfaces.C.int;
subtype Constrained_List is List (1 .. 100);
procedure P (V : Constrained_List; Size : Interfaces.C.int);
pragma Import (C, P, "p");
Obj : Constrained_List := (others => 42); -- arbitrary values
```

With that, we can just pass the value by reference as usual on the C side:

```
void p (int* v, int size) {
    // whatever
}
```

But that's assuming we know how many array components are sufficient from the C code's point of view. In the example above we'll pass a value up to 100 to the `Size` parameter and hope that is sufficient.

Really, it would work to use the unconstrained array type as the formal parameter type instead:

```
type List is array (Integer range <>) of Interfaces.C.int;
procedure P (V : List; Size : Interfaces.C.int);
pragma Import (C, P, "p");
```

The C function parameter profile wouldn't change. But why does this work? With values of unconstrained array types, the bounds are stored with the value. Typically they are stored just ahead of the first component, but it is implementation-defined. So why doesn't the above accidentally pass the bounds instead of the first array component itself? It works because we are guaranteed by the Ada language that passing an array will pass (the address of) the components, not the bounds, even for Ada unconstrained array types.

Now for the other direction: passing an array from C to Ada. Here the lack of bounds information on the C side really makes a difference. We can't just pass the array by itself because that would not include the bounds, unlike an Ada call to an Ada routine. In this case the approach is the similar to the first alternative described above, in which we declare a very large array and then pass the bounds explicitly:

```
type List is array (Natural) of int;
-- DO NOT DECLARE AN OBJECT OF THIS TYPE

procedure P (V : List; Size : Interfaces.C.int);
pragma Export (C, P, "p");

procedure P (V : List; Size : Interfaces.C.int) is
begin
    for J in 0 .. Size - 1 loop
        -- whatever
    end loop;
end P;
```

```
extern void p (int* v, int size);

int x [100];

p (x, 100); // call to Ada routine, passing x
```

The fundamental idea is to declare an Ada type big enough to handle anything conceivably needed on the C side. Subtype **Natural** means `0 .. Integer'Last` so `List` is quite large indeed. Just be sure never to declare an object of that type. You'll probably run out of storage on an embedded target.

Earlier we said that it is the Ada type that determines how parameters are passed, and that scalars and elementary types are always passed by copy. For mode `in` that's simple, the copy to the C formal parameter is done and that's all there is to it. But suppose the mode is instead `out` or `in out`? In that case the presumably updated value must be returned to the caller, but C doesn't do that by copy. Here the compiler will come to the rescue and make it work, transparently. Specifically, we just declare the Ada subprogram's formal parameter type as usual, but on the C formal we use a reference. We're talking about scalar and elementary types so let's use `int` arbitrarily. We make the mode `in out` but `out` would also serve:

```
procedure P (Formal : in out int);

void function p (int* formal);
```

Now the compiler does its magic: it generates code to make a copy of the actual parameter, but it makes that copy into a hidden temporary object. Then, when calling the C routine, it passes the address of the hidden object, which corresponds to the reference expected on the C side. The C code updates the value of the temporary object via the reference, and then, on return, the compiler copies the value back from the temporary to the actual parameter. Problem solved, if a bit circuitous.

There are other aspects to interfacing with C, such as variadic functions that take a varying number of arguments, but you can find these elsewhere in the learn courses.

Next, we examine the child packages under `Interfaces.C`. These packages are not used as much as the parent `Interfaces.C` package so we will provide an overview. You can look up the contents within GNAT Studio or the Ada language standard.

32.2.2 Package Interfaces.C.Strings

Package `Interfaces.C` declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type `chars_ptr` corresponds to a common use of `char *` in C programs, and an object of this type can be passed to imported subprograms for which `char *` is the type of the argument of the C function. A subset of the package content is as follows:

```
package Interfaces.C.Strings is

    type chars_ptr is private;
    ...

    function New_Char_Array (Chars : in char_array) return chars_ptr;
    function New_String (Str : in String) return chars_ptr;
    procedure Free (Item : in out chars_ptr);
```

(continues on next page)

(continued from previous page)

```

...
function Value (Item : in chars_ptr) return char_array;
function Value (Item : in chars_ptr) return String;
...

function Strlen (Item : in chars_ptr) return size_t;

procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Chars  : in char_array;
                  Check   : in Boolean := True);

...
end Interfaces.C.Strings;

```

Note that allocation might be via malloc, or via Ada's allocator `new`. In either case, the returned value is guaranteed to be compatible with `char*`. Deallocation must be via the supplied procedure `Free`.

An amusing point is that you can overwrite the end of the char array just like you can in C, via procedure `Update`. The `Check` parameter indicates whether overwriting past the end is checked. The default is `True`, unlike in C, but you could pass an explicit `False` if you felt the need to do something questionable.

32.2.3 Package `Interfaces.C.Pointers`

The generic package `Interfaces.C.Pointers` allows us to perform C-style operations on pointers. It includes an access type named `Pointer`, various `Value` functions that dereference a `Pointer` value and deliver the designated array, several pointer arithmetic operations, and "copy" procedures that copy the contents of a source pointer into the array designated by a destination pointer.

We won't go into the details further. See the Ada RM for more.

32.2.4 Package `Interfaces.Fortran`

Like `Interfaces.C`, package `Interfaces.Fortran` defines Ada types to be used when working with subprograms using the Fortran calling convention. These types have representations that are identical to the default representations of the Fortran intrinsic types `Integer`, `Real`, `Double Precision`, `Complex`, `Logical`, and `Character` in some supported Fortran implementation. And like the C package, the ways that parameters of various types are passed are also specified.

We leave the details to you to look up in the language standard, if you find them needed in an embedded application.

32.2.5 Machine Code Insertions (MCI)

When working close to the hardware, especially when interacting with a device, it is not uncommon for the hardware to require a very specific set of assembly language instructions to be generated. There are two ways to achieve this: the right way and the wrong way.

The wrong way is to experiment with the source code and compiler switches until you get the exact assembly code you need generated (assuming it is possible at all). But what happens when the next compiler release arrives with a new optimization? And abandon all hope if you go to a new compiler vendor. This approach is both labor-intensive and very brittle.

The right way is to express the precise assembly code sequence explicitly within the Ada source code. (That's true to any high level language, not just Ada.) Or you can call an intrinsic function, if there is one that does exactly what you need. We will focus on inserting it directly, in what is known as "machine code insertion", or "inline assembler."

As an example of the need for this capability, consider the GPIO (General Purpose I/O) port on an STM32 Arm microcontroller. Each port contains 16 individual I/O pins, each of which can be configured as an independent discrete input or output, or as a control line for a device, with pull-up or pull-down registers, with different clock speeds, and so on. Different on-chip devices use various collections of pins in ways specific to the devices, and require exclusive assignment of the pins. However, any given pin can be used by several different devices. For example, pin 11 on port A ("PA11") can be used by USART #1 as the clear-to-send ("CTS") line, or the CAN #1 bus Rx line, or Channel 4 of Timer 1, among others. Therefore, one of the responsibilities of the system designer is to allocate pins to devices, ensuring that they are allocated uniquely. It is difficult to debug the case in which a pin is accidentally configured for one device and then reconfigured for use with another device (assuming the first device remains in use). To help ensure exclusive allocations, every GPIO port on this Arm implementation has a way of locking the configuration of each I/O pin. That way, some other part of the software can't successfully change the configuration accidentally, for use with some other device. Even if the same configuration was to be used for another device, the lock prevents the accidental update so we find out about the unintentional sharing.

To lock a pin on a port requires a special sequence of reads and writes to a GPIO register for that port. A specific bit pattern is required during the reads and writes. The sequence and bit pattern is such that accidentally locking the pin is highly unlikely.

Once we see how to express assembly language sequences in general we will see how to get the necessary sequence to lock a port/pin pair. Unfortunately, although you can express exactly the code sequence required, such a sequence of assembly language instructions is clearly target hardware-specific. That means portability is inherently limited. Moreover, the syntax for expressing it varies with the vendor, even for the same target hardware. Being able to insert it at the Ada source level doesn't help with either portability issue. You should understand that the use-case for machine code insertion is for small, short sequences. Otherwise you would write the code in assembly language directly, in a separate file. That might obtain a degree of vendor independence, at least for the given target, but not necessarily. The use of inline assembler is intended for cases in which a separate file containing assembly language is not simpler.

With those caveats in place, let's first examine how to do it in general and then how to express it with GNAT specifically.

The right way to express an arbitrary sequence of one or more assembly language statements is to use so-called "code statements." A code statement is an Ada statement, but it is also a qualified expression of a type defined in package `System.Machine_Code`. The content of that package, and the details of code statements, are implementation-defined. Although that affects portability there really is no alternative because we are talking about machine instruction sets, which vary considerably and cannot be standardized at this level.

Package `System.Machine_Code` contains types whose values provide a way of expressing

assembly instructions. For example, let's say that there is a "HLT" instruction that halts the processor for some target. There is no other parameter required, just that op-code. Let's also say that one of the types in System.Machine_Code is for these "short" instructions consisting only of an op-code. The syntax for the type declaration would then allow the following code statement:

```
Short_Instruction'(Command => HLT);
```

Each of Short_Instruction, Command, and HLT are defined by the vendor in this hypothetical version of package System.Machine_Code. You can see why we say that it is both a statement (note the semicolon) and a qualified expression (note the apostrophe).

Code statements must appear in a subprogram body, after the **begin**. Only code statements are allowed in such a body, only use-clauses can be in the declarative part, and no exception handlers are allowed. The complete example would be as follows:

```
procedure Halt -- stops processor
  with Inline;

with System.Machine_Code; use System.Machine_Code;
procedure Halt is
begin
  Short_Instruction'(Command => HLT);
end Halt;
```

With that, to halt the processor the Ada code can simply call procedure Halt. When the optimizer is enabled there will be no code emitted to make the call, we'd simply see the halt instruction emitted directly in-line.

Package System.Machine_Code provides access to machine instructions but as we mentioned, the content is vendor-defined. In addition, the package itself is optional, but is required if Annex C, the Systems Programming Annex, is implemented by the vendor. In practice most all vendors provide this annex.

In GNAT, the content of System.Machine_Code looks something like this:

```
type Asm_Input_Operand is ...
type Asm_Output_Operand is ...
type Asm_Input_Operand_List is array (Integer range <>) of Asm_Input_Operand;
type Asm_Output_Operand_List is array (Integer range <>) of Asm_Output_Operand;

type AsmInsn is private;

...

function Asm
  (Template : String;
   Outputs : Asm_Output_Operand := No_Output_Operands;
   Inputs  : Asm_Input_Operand  := No_Input_Operands;
   Clobber : String  := "";
   Volatile : Boolean := False) return AsmInsn;
```

With this package content, the expression in a code statement is of type AsmInsn, short for "assembly instruction." Multiple overloaded functions named Asm return values of that type.

The Template parameter in a string containing one or more assembly language instructions. These instructions are specific to the target machine. The parameter Outputs provides mappings from registers to source-level entities that are updated by the assembly statement(s). Inputs provides mappings from source-level entities to registers for inputs. Volatile, when True, tells the compiler not to optimize the call away, and Clobber tells the compiler which registers, or memory, if any, are altered by the instructions in Template.

("Clobber" is colloquial English for "destroy.") That last is important because the compiler was likely already using some of those registers so the compiler will need to restore them after the call.

We could say, for example, the following, taking all the defaults except for Volatile:

```
Asm ("nop", Volatile => True);
```

As you can imagine the full details are extensive, beyond the scope of this introduction. See the GNAT User Guide ("Inline Assembler") for all the gory details.

Now, back to our GPIO port/bin locking example. The port type is declared as follows:

```
type GPIO_Port is limited record
  ...
  LCKR : Word with Atomic; -- lock register
  ...
end record with ...
```

We've elided all but the LCKR component representing the "lock register" within each port. We'd have a record representation clause to ensure the required layout but that's not important here. Word is an unsigned (modular) 32-bit integer type. One of the hardware requirements for accessing the lock register is that the entire register has to be read or written whenever any bits within it are accessed. The compiler must not, for example, write one of the bytes within the register in order to set or clear a bit within that part of the register. Therefore we mark the register as Atomic. If the compiler cannot honor that aspect the compilation will fail, so we would know there is a problem.

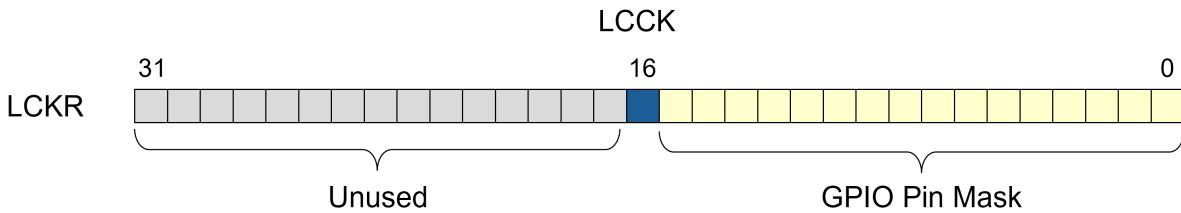
Per the ST Micro Reference Manual, the lock control bit is referred to as LCKK and is bit #16, i.e., the first in the upper half of the LCKR register word.

```
LCKK : constant Word := 16#0001_0000#; -- the "lock control bit"
```

That bit is also known as the "Lock Key" (hence the abbreviation) because it is used to control the locking of port/pin configurations.

There are 16 GPIO pins per port, represented by the lower 16 bits of the register. Each one of these 16 bits corresponds to one of the 16 GPIO pins on a port. If any given bit reads as a 1 then the corresponding pin is locked.

Graphically that looks like this:



Therefore, the Ada types are:

```
type GPIO_Pin is
  (Pin_0, Pin_1, Pin_2, Pin_3, Pin_4, Pin_5, Pin_6, Pin_7,
   Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);

for GPIO_Pin use (Pin_0  => 16#0001#,
                  Pin_1  => 16#0002#,
                  Pin_2  => 16#0004#,
                  ...
                  Pin_15 => 16#8000#);
```

Note that we had to override the default enumeration representation so that each pin — each enumeral value — would occupy a single dedicated bit in the bit-mask.

With that in place, let's lock a pin. A specific sequence is required to set a pin's lock bit. The sequence writes and reads values from the port's LCKR register. Remember that this 32-bit register has 16 bits for the pin mask (0 .. 15), with bit #16 used as the "lock control bit".

1. write a 1 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
2. write a 0 to the lock control bit with a 1 in the pin bit mask for the pin to be locked
3. do step 1 again
4. read the entire LCKR register
5. read the entire LCKR register again (optional)

Throughout the sequence the same value for the lower 16 bits of the word must be maintained (i.e., the pin mask), including when clearing the LCCK bit in the upper half.

If we wrote this in Ada it would look like this:

```
procedure Lock (Port : in out GPIO_Port; Pin : GPIO_Pin) is
  Temp : Word with Volatile;
begin
  -- set the lock control bit and the pin bit, clear the others
  Temp := LCCK or Pin'Enum_Rep;
  -- write the lock and pin bits
  Port.LCKR := Temp;
  -- clear the lock bit in the upper half
  Port.LCKR := Pin'Enum_Rep;
  -- write the lock bit again
  Port.LCKR := Temp;
  -- read the lock bit
  Temp := Port.LCKR;
  -- read the lock bit again
  Temp := Port.LCKR;
end Lock;
```

Pin'Enum_Rep gives us the underlying value for the enumeration value. We cannot use 'Pos because that attribute provides the logical position number within the enumerated values, and as such always increases consecutively. We need the underlying representation value that we specified explicitly.

The Ada procedure works, but only if the optimizer is enabled (which also precludes debugging). But even so, there is no guarantee that the required assembly language instruction sequence would be generated, especially one that maintains that required bit mask value on each access. A machine-code insertion is appropriate for all the reasons presented earlier:

```
procedure Lock (Port : in out GPIO_Port;
                 Pin : GPIO_Pin) is
  use System.Machine_Code, ASCII, System;
begin
  Asm ("orr r3, %1, #65536" & LF & HT & -- 0) Temp := LCCK or Pin'Enum_Rep
  "str r3, [%0, #28]" & LF & HT & -- 1) Port.LCKR := Temp
  "str %1, [%0, #28]" & LF & HT & -- 2) Port.LCKR := Pin'Enum_Rep
  "str r3, [%0, #28]" & LF & HT & -- 3) Port.LCKR := Temp
  "ldr r3, [%0, #28]" & LF & HT & -- 4) Temp := Port.LCKR
  "ldr r3, [%0, #28]" & LF & HT, -- 5) Temp := Port.LCKR
  Inputs => (Address'Asm_Input ("r", This'Address), -- %0
              (GPIO_Pin'Asm_Input ("r", Pin))), -- %1
  Volatile => True,
```

(continues on next page)

(continued from previous page)

```
Clobber  => ("r3"));
end Lock;
```

We've combined the instructions into one Asm expression. As a result, we can use ASCII line-feed and horizontal tab characters to format the listing produced by the compiler so that each instruction is on a separate line and aligned with the previous instruction, as if we had written the sequence in assembly language directly. That enhances readability later, during examination of the compiler output to verify the required sequence was emitted.

In the above, "%0" is the first input, containing the address of the Port parameter. "%1" is the other input, the value of the Pin parameter. We're using register r3 explicitly, as the "temporary" variable, so we tell the compiler that it has been "clobbered."

If we examine the assembly language output from compiling the file, we find the body of procedure Lock is as hoped:

```
ldr  r2, [r0, #4]
ldrh r1, [r0, #8]
.syntax unified
orr  r3, r1, #65536
str  r3, [r2, #28]
str  r1, [r2, #28]
str  r3, [r2, #28]
ldr  r3, [r2, #28]
ldr  r3, [r2, #28]
```

The first two statements load register 2 (r2) and register 1 (r1) with the subprogram parameters, i.e., the port and pin, respectively. Register 2 gets the starting address of the port record, in particular. (Offset #28 is the location of the LCKR register. The port is passed by reference so that address is actually that of the hardware device.)

We will have separately declared procedure Lock with inlining enabled, so whenever we call the procedure we will get the exact assembly language sequence required to lock the indicated pin on the given port, without any additional code for a procedure call.

Note that we get the calling convention right automatically, because the subprogram is not a foreign entity written in some other language (such as assembly language). It's an Ada subprogram with special content so the Ada convention applies as usual.

32.3 When Ada Is Not the Main Language

When multiple programming languages are involved, the main procedure might not be implemented in Ada. Maybe the bulk of the program is written in C, for example, and this C code calls some Ada routines that have been exported (with the C convention).

That means the Ada builder does not create the executable image's entry point. In fact the Ada main procedure is never the entry point for the final executable image, it's just where the application code begins, like the C main function. There are setup and initialization steps that must happen before any program can execute on a target, and the entry point code is responsible for this functionality. For example, on a bare machine target, the hardware must be initialized, the trap vectors installed, the segments initialized, and so on. On a target running an operating system, the OS is responsible for that initialization but there will be OS-specific initialization steps too. For example, if command-line arguments are supported these may be gathered. All this initialization code is generated by the builder, regardless of the language, followed by a call to the main routine.

Some of the initialization is specific to Ada programming, and must occur before any calls occur to the exported Ada routines. In particular, the entry point code emitted by the Ada builder initializes the Ada run-time system and calls all the elaboration routines for the

library units in the application code. Only then does the emitted code invoke the Ada main. If the Ada builder is not going to create the executable it has no chance to emit the code to do that prior initialization. A foreign language builder will not emit such code, so we have a problem.

You could learn enough about how the foreign builder works, and how your Ada builder works, to create a work-around. You could learn what the Ada builder would emit, in other words, and ensure those routines are called manually, either directly or by augmenting the builder scripts (assuming that's possible). But the work-around would be labor-intensive and not robust to changes by the tool vendors. It would be an ugly hack, in other words.

That work-around would not be portable either. The Ada standard can't address hardware- or OS-specific initialization, but it can standardize the name for a routine to do the Ada-specific initialization. Specifically, procedure `adainit` initializes the Ada application code and the Ada run-time library. Similarly, one might need to shut down the Ada code when no further calls will be made to the exported Ada routines. Procedure `adafinal` performs this shut-down functionality. Neither procedure has parameters.

The main function in the other language is intended to import these routines and manually call them each exactly once. `adainit` must be called prior to any calls to the Ada code, and `adafinal` is to be called after all the calls to the Ada code.

For example:

```
#include "stdio.h"

extern int checksum (char *input, int count);

extern void adainit (void);
extern void adafinal (void);

int main (int argc, char *argv[]) {
    char * Str = "Hello World!";
    int sum;
    adainit ();
    sum = checksum (Str, strlen (Str));
    adafinal ();
    printf ("checksum for '%s' is %d", Str, sum);
    return 0;
}
```

In the above, we have an Ada routine to compute a checksum, called by a C main function. Therefore, we use "extern" to tell the C compiler that the "checksum" function is defined elsewhere, i.e., in the Ada routine. Likewise, we tell the compiler that functions `adainit` and `adafinal` are defined elsewhere. The call to `adainit` is made before the call to any Ada code, thus all the elaboration code is guaranteed to happen before `checksum` needs it. Once the Ada code is not needed, the call to `adafinal` can be made.

Both `adainit` and `adafinal` have no effect after the first invocation. That means you cannot structure your foreign code to iteratively call the two routines whenever you want to invoke some Ada code. In practice you just call them once in the main and be done with it.

CHAPTER
THIRTYTHREE

INTERACTING WITH DEVICES

Interacting with hardware devices is one of the more frequent activities in embedded systems programming. It is also one of the most enjoyable because you can make something happen in the physical world. There's a reason that making an LED blink is the "hello world" of embedded programming. Not only is it easy to do, it is surprisingly satisfying. I suspect that even the developers of "Full Authority Digital Engine Controllers" (FADEC) — the computers that are in complete, total control of commercial airline engines — have fond memories of making an LED blink early in their careers. And of course a blinking LED is a good way to indicate application status, especially if off-board I/O is limited, which is often the case.

Working at the device register level can be error prone and relatively slow, in terms of source-lines-of-code (SLOC) produced. That's partly because the hardware is in some cases complicated, and partly because of the way the software is written. Using bit masks for setting and clearing bits is not a readable approach, comparatively speaking. There's just not enough information transmitted to the reader. It might be clear enough when written, but will you see it that way months later? Readability is important because programs are read many more times than they are written. Also, an unreadable program is more difficult to maintain, and maintenance is where most money is spent in long-lived applications. Comments can help, until they are out of date. Then they are an active hindrance.

For example, what do you think the following code does? This is real code, where `temp` and `temp2` are unsigned 32-bit integers:

```
temp = ((uint32_t)(GPIO_AF) <<
          ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
          ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
```

That's unfair to ask, absent any context. The code configures a general purpose I/O (GPIO) pin on an Arm microcontroller for one of the "alternate functions". `GPIOx` is a pointer to a GPIO port, `GPIO_PinSource` is a GPIO pin number, and `GPIO_AF` is the alternate function number. But let's say you knew that. Is the code correct? The longer it takes to know, the less productive you are.

The fact that the code above is in C is beside the point. If we wrote it the same way in Ada it would be equally opaque, if not more so. There are simpler approaches. Judicious use of record and array types is one. We'll say more about that later, but the underlying idea is to let the compiler do as much work for us as possible. For example, the data structures used in the code above require explicit shifting whenever they are accessed. If we can avoid that at the source code level — by having the compiler do it for us — we will have simplified the code considerably. Furthermore, letting the compiler do the work for us makes the code more maintainable (which is where the money is). For example, if the code does the shifting explicitly and the data structures are changed, we'll have to change the number of bits to shift left or right. Constants will help there, but we still have to remember to change them;

the compiler won't complain if we forget. In contrast, if we let the compiler do this shifting for us, the amounts to shift will be changed automatically.

Some devices are very simple. In these cases the application may interact directly with the device without unduly affecting productivity. For example, there was a board that had a user-accessible rotary switch with sixteen distinct positions. Users could set the switch to whatever the application code required, e.g., to indicate some configuration information. The entire software interface to this device consisted of a single read-only 8-bit byte in memory. That's all there was to it: you read the memory and thus got the numeric setting of the switch.

More complex devices, however, usually rely on software abstraction to deal with the complexity. Just as abstraction is a fundamental way to combat complexity in software, abstraction also can be used to combat the complexity of driving sophisticated hardware. The abstraction is presented to users by a software "device driver" that exists as a layer between the application code and the hardware device. The layer hides the gory details of the hardware manipulation behind subprograms, types, and parameters.

We say that the device driver layer is an abstraction because, at the least, the names of the procedures and functions indicate what they do, so at the call site you can tell *what* is being done. That's the point of abstraction: it allows us to focus on what, rather than how. Consider that GPIO pin configuration code block again. Instead of writing that block every time we need to configure the alternate function for a pin, suppose we called a function:

```
GPIO_PinAFConfig(USARTx_TX_GPIO_PORT, USARTx_TX_SOURCE, USARTx_TX_AF);
```

The `GPIO_PinAFConfig` function is part of the GPIO device driver provided by the STM32 Standard Peripherals Library (SPL). Even though that's not the best function name conceivable, calls to the function will be far more readable than the code of the body, and we only have to make sure the function implementation is correct once. And assuming the device drivers' subprograms can be inlined, the subprogram call imposes no performance penalty.

Note the first parameter to the call above: `USARTx_TX_GPIO_PORT`. There are multiple GPIO ports on an Arm implementation; the vendor decides how many. In this case one of them has been connected to a USART (Universal Synchronous Asynchronous Receiver Transmitter), an external device for sending and receiving serial data. When there are multiple devices, good software engineering suggests that the device driver present a given device as one of a type. That's what an "abstract data type" (ADT) provides for software and so the device driver applies the same design. An ADT is essentially a class, in class-oriented languages. In Ada, an ADT is represented as a private type declared in a package, along with subprograms that take the type as a parameter.

The Ada Drivers Library (ADL) provided by AdaCore and the Ada community uses this design to supply Ada drivers for the timers, I2C, A/D and D/A converters, and other devices common to microcontrollers. Multiple devices are presented as instances of abstract data types. A variety of development platforms from various vendors are supported, including the STM32 series boards. The library is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library. We are going to use some of these drivers as illustrations in the following sections.

33.1 Non-Memory-Mapped Devices

Some devices are connected to the processor on a dedicated bus that is separate from the memory bus. The Intel processors, for example, used to have (and may still have) instructions for sending and receiving data on this bus. These are the "in" and "out" instructions, and their data-length specific variants.

The original version of Ada defined a package named `Low_Level_I0` for such architectures, but there were very few implementations (maybe just one, known to support the Intel processors). As a result, the package was actually removed from the language standard. Implementations could still support the package, it just wouldn't be a standard package. That's different from constructs that are marked as "obsolescent" by the standard, e.g., the pragmas replaced by aspects, among other things. Obsolescent constructs are still part of the standard.

If a given target machine has such I/O instructions for the device bus, these can be invoked in Ada via machine-code insertions. For example:

```
procedure Send_Control (Device : Port; Data : Unsigned_16) is
    pragma Suppress (All_Checks);
begin
    asm ("outw %1, (%0)",
         Inputs  => (Port'Asm_Input("dx",Device),
                       Unsigned_16'Asm_Input("ax",Data)),
         Clobber => "ax, dx");
end Send_Control;

procedure Receive_Control (Device : Port; Data : out Unsigned_16) is
    pragma Suppress (All_Checks);
begin
    asm ("inw (%1), %0",
         Inputs   => (Port'Asm_Input("dx",Device)),
         Outputs  => (Unsigned_16'Asm_Output("=ax",Data)),
         Clobber  => "ax, dx",
         Volatile => True);
end Receive_Control;
```

Applications could use these subprograms to set the frequency of the Intel PC tone generator, for example, and to turn it on and off. (You can't do that any more in application code because modern operating systems don't give applications direct access to the hardware, at least not by default.)

Although the `Low_Level_I0` package is no longer part of the language, you can write this sort of thing yourself, or vendors can do it. That's possible because the Systems Programming Annex, when implemented, guarantees fully effective use of machine-code inserts. That means you can express anything the compiler could emit. The guarantee is important because otherwise the compiler might "get in the way." For example, absent the guarantee, the compiler would be allowed to insert additional assembly language statements in between yours. That can be a real problem, depending on what your statements do. For instance, if your MCI assembly statements do something and then check a resulting condition code, such as the overflow flag, those interleaved compiler-injected statements might clear that condition code before your code can check it. Fortunately, the annex guarantees that sort of thing cannot happen.

33.2 Memory-Mapped Devices

In *another earlier chapter* (page 399), we said that we could query the address of some object, and we also showed how to use that result to specify the address of some other object. We used that capability to create an "overlay," in which two objects are used to refer to the same memory locations. As we indicated in that discussion, you would not use the same type for each object — the point, after all, is to provide a view of the shared underlying memory cells that is not already available otherwise. Each distinct type would provide a distinct view of the memory values, that is, a set of operations providing some required functionality.

For example, here's an overlay composed of a 32-bit signed integer object and a 32-bit array object:

```
type Bits32 is array (0 .. 31) of Boolean
  with Component_Size => 1;

X : aliased Integer_32;
Y : Bits32 with Address => X'Address;
```

Because one view is as an integer and the other as an array, we can access that memory using the two different views' operations. Using the view as an array object (Y) we can access individual bits of the memory shared with X. Using the view as an integer (X), we can do arithmetic on the contents of that memory. (We could have used an unsigned integer instead of the signed type, and thereby gained the bit-oriented operations, but that's not the point.)

Very often, though, there is only one Ada object that we place at some specific address. That's because the Ada object is meant to be the software interface to some memory-mapped hardware device. In this scenario we don't have two overlaid Ada objects, we just have one. The other "object" is the hardware device mapped to that starting address. Since they are at the same memory location(s), accessing the Ada object accesses the hardware device.

For a real-world but nonetheless simple example, recall that example of a rotary switch on the front of our embedded computer that we mentioned in the introduction. This switch allows humans to provide some very simple input to the software running on the computer.

```
Rotary_Switch : Unsigned_8 with
  Address => System.Storage_Elements.To_Address (16#FFC0_0801#);
```

We declare the object and also specify the address, but not by querying some entity. We already know the address from the hardware documentation. But we cannot simply use an integer address literal from that documentation because type `System.Address` is almost always a private type. We need a way to compose an `Address` value from an integer value. The package `System.Storage_Elements` defines an integer representation for `Address` values, among other useful things, and a way to convert those integer values to `Address` values. The function `To_Address` does that conversion.

As a result, in the Ada code, reading the value of the variable `Rotary_Switch` reads the number on the actual hardware switch.

Note that if you specify the wrong address, it is hard to say what happens. Likewise, it is an error for an address clause to disobey the object's alignment. The error cannot be detected at compile time, in general, because the address is not necessarily known at compile time. There's no requirement for a run-time check for the sake of efficiency, since efficiency seems paramount here. Consequently, this misuse of address clauses is just like any other misuse of address clauses — execution of the code is erroneous, meaning all bets are off. You need to know what you're doing.

What about writing to the variable? Is that meaningful? In this particular example, no. It

is effectively read-only memory. But for some other device it very well could be meaningful, certainly. It depends on the hardware. But in this case, assigning a value to the `Rotary_Switch` variable would have no effect, which could be confusing to programmers. It looks like a variable, after all. We wouldn't declare it as a constant because the human user could rotate the switch, resulting in a different value read. Therefore, we would hide the Ada variable behind a function, precluding the entire issue. Clients of the function can then use it for whatever purpose they require, e.g., as the unique identifier for a computer in a rack.

Let's talk more about the type we use to represent a memory-mapped device. As we said, that type defines the view we have for the object, and hence the operations we have available for accessing the underlying mapped device.

We choose the type for the representative Ada variable based on the interface of the hardware mapped to the memory. If the interface is a single monolithic register, for example, then an integer (signed or unsigned) of the necessary size will suffice. But suppose the interface is several bytes wide, and some of the bytes have different purposes from the others? In that case, a record type is the obvious solution, with distinct record components dedicated to the different parts of the hardware interface. We could use individual bits too, of course, if that's what the hardware does. Ada is particularly good at this fine-degree of representation because record components of any types can be specified in the layout, down to the bit level, within the record.

In addition, we might want to apply more than one type, at any one time, to a given memory-mapped device. Doing so allows the client code some flexibility, or it might facilitate an internal implementation. For example, the STM32 boards from ST Microelectronics include a 96-bit device unique identifier on each board. The identifier starts at a fixed memory location. In this example we provide two different views — types — for the value. One type provides the identifier as a `String` containing twelve characters, whereas another type provides the value as an array of three 32-bit unsigned words (i.e., 12 bytes). The two types are applied by two overloaded functions that are distinguished by their return type:

```
package STM32.Device_Id is

    subtype Device_Id_Image is String (1 .. 12);

    function Unique_Id return Device_Id_Image;

    type Device_Id_Tuple is array (1 .. 3) of UInt32
        with Component_Size => 32;

    function Unique_Id return Device_Id_Tuple;

end STM32.Device_Id;
```

The subtype `Device_Id_Image` is the view of the 96-bits as an array of twelve 8-bit characters. (Using type `String` here isn't essential. We could have defined an array of bytes instead of `Character`.) Similarly, subtype `Device_Id_Tuple` is the view of the 96-bits as an array of three 32-bit unsigned integers. Clients can then choose how they want to view the unique id by choosing which function to call.

In the package body we implement the functions as two ways to access the same shared memory:

```
with System;

package body STM32.Device_Id is

    ID_Address : constant System.Address := System'To_Address (16#1FFF_7A10#);

    function Unique_Id return Device_Id_Image is
```

(continues on next page)

(continued from previous page)

```

Result : Device_Id_Image with Address => ID_Address, Import;
begin
    return Result;
end Unique_Id;

function Unique_Id return Device_Id_Tuple is
    Result : Device_Id_Tuple with Address => ID_Address, Import;
begin
    return Result;
end Unique_Id;

end STM32.Device_Id;

```

The GNAT-defined attribute System'To_Address in the declaration of ID_Address is the same as the function System.Storage_Elements.To_Address except that, if the argument is static, the function result is static. 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 (because the function call is always non-static, even if its argument is static).

The only difference in the bodies is the return type and matching type for the local Result variable. Both functions read from the same location in memory.

Earlier we indicated that the bit-pattern implementation of the GPIO function could be expressed differently, resulting in more readable, therefore maintainable, code. The fact that the code is in C is irrelevant; the same approach in Ada would not be any better. Here's the complete code for the function body:

```

void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t      GPIO_PinSource,
                      uint8_t       GPIO_AF)
{
    uint32_t temp = 0x00;
    uint32_t temp_2 = 0x00;

    /* Check the parameters */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
    assert_param(IS_GPIO_AF(GPIO_AF));

    temp = ((uint32_t)(GPIO_AF) <<
              ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
    GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
          ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
    temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
    GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}

```

The problem, other than the magic numbers (some named constants would have helped), is that the code is doing nearly all the work instead of off-loading it to the compiler. Partly that's because in C we cannot declare a numeric type representing a 4-bit quantity, so everything is done in terms of machine units, in this case 32-bit unsigned integers.

Why do we need 4-bit values? At the hardware level, each memory-mapped GPIO port has a sequence of 16 4-bit quantities, one for each of the 16 pins on the port. Those 4-bit quantities specify the "alternate functions" that the pin can take on, if needed. The alternate functions allow a given pin to do more than act as a single discrete I/O pin. For example, a pin could be connected to the incoming lines of a USART. We use the configuration routine to apply the specific 4-bit code representing the alternate function required for our application.

These 16 4-bit alternate function fields are contiguous in the register (hence memory) so we can represent them as an array with a total size of 64-bits (i.e., 16 times 4). In the C

version this array has two components of type `uint32_t` so it must compute where the corresponding 4-bit value for the pin is located within those two words. In contrast, the Ada version of the array has components of the 4-bit type, rather than two 32-bit components, and simply uses the pin number as the index. The resulting Ada procedure body is extremely simple:

```
procedure Configure_Alternate_Function
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin;
   AF   : GPIO_Alternate_Function_Code)
is
begin
  Port.AFR (Pin) := AF;
end Configure_Alternate_Function;
```

In the Ada version, AFR is a component within the `GPIO_Port` record type, much like in the C code's struct. However, Ada allows us to declare a much more descriptive set of types, and it is these types that allows the developer to off-load the work to the compiler.

First, in Ada we can declare a 4-bit numeric type:

```
type Bits_4 is mod 2**4 with Size => 4;
```

The `Bits_4` type was already globally defined elsewhere so we just derive our 4-bit "alternate function code" type from it. Doing so allows the compiler to enforce simple strong typing so that the two value spaces are not accidentally mixed. This approach also increases understanding for the reader:

```
type GPIO_Alternate_Function_Code is new Bits_4;
-- We cannot use an enumeration type because there are duplicate binary
-- values
```

Hence type `GPIO_Alternate_Function_Code` is a copy of `Bits_4` in terms of operations and values, but is not the same type as `Bits_4` so the compiler will keep them separate for us.

We can then use that type as the array component type for the representation of the AFR:

```
type Alternate_Function_Fields is
  array (GPIO_Pin) of GPIO_Alternate_Function_Code
  with Component_Size => 4, Size => 64; -- both in units of bits
```

Note that we can use the `GPIO_Pin` parameter directly as the index into the array type, obviating any need to massage the `Pin` value in the procedure. That's possible because the type `GPIO_Pin` is an enumeration type:

```
type GPIO_Pin is
  (Pin_0, Pin_1, Pin_2, Pin_3, Pin_4, Pin_5, Pin_6, Pin_7,
   Pin_8, Pin_9, Pin_10, Pin_11, Pin_12, Pin_13, Pin_14, Pin_15);

for GPIO_Pin use
  (Pin_0  => 16#0001#,
   Pin_1  => 16#0002#,
   Pin_2  => 16#0004#,
   Pin_3  => 16#0008#,
   Pin_4  => 16#0010#,
   Pin_5  => 16#0020#,
   Pin_6  => 16#0040#,
   Pin_7  => 16#0080#,
   Pin_8  => 16#0100#,
   Pin_9  => 16#0200#,
   Pin_10 => 16#0400#,
```

(continues on next page)

(continued from previous page)

```
Pin_11 => 16#0800#,
Pin_12 => 16#1000#,
Pin_13 => 16#2000#,
Pin_14 => 16#4000#,
Pin_15 => 16#8000#);
```

In the hardware, the GPIO_Pin values don't start at zero and monotonically increase. Instead, the values are bit patterns, where one bit within each value is used. The enumeration representation clause allows us to express that representation.

Type Alternate_Function_Fields is then used to declare the AFR record component in the GPIO_Port record type:

```
type GPIO_Port is limited record
    MODER      : Pin_Modes_Register;
    OTYPER     : Output_Types_Register;
    Reserved_1 : Half_Word;
    OSPEEDR   : Output_Speeds_Register;
    PUPDR     : Resistors_Register;
    IDR        : Half_Word;          -- input data register
    Reserved_2 : Half_Word;
    ODR        : Half_Word;          -- output data register
    Reserved_3 : Half_Word;
    BSRR_Set   : Half_Word;          -- bit set register
    BSRR_Reset : Half_Word;          -- bit reset register
    LCKR       : Word with Atomic;
    AFR        : Alternate_Function_Fields;
    Unused     : Unaccessed_Gap;
end record with
Size => 16#400# * 8;

for GPIO_Port use record
    MODER      at 0 range 0 .. 31;
    OTYPER     at 4 range 0 .. 15;
    Reserved_1 at 6 range 0 .. 15;
    OSPEEDR   at 8 range 0 .. 31;
    PUPDR     at 12 range 0 .. 31;
    IDR        at 16 range 0 .. 15;
    Reserved_2 at 18 range 0 .. 15;
    ODR        at 20 range 0 .. 15;
    Reserved_3 at 22 range 0 .. 15;
    BSRR_Set   at 24 range 0 .. 15;
    BSRR_Reset at 26 range 0 .. 15;
    LCKR       at 28 range 0 .. 31;
    AFR        at 32 range 0 .. 63;
    Unused     at 40 range 0 .. 7871;
end record;
```

These declarations define a record type that matches the content and layout of the STM32 GPIO Port memory-mapped device.

Let's compare the two procedure implementations again. Here they are, for convenience:

```
void GPIO_PinAFConfig(GPIO_TypeDef *GPIOx,
                      uint16_t      GPIO_PinSource,
                      uint8_t       GPIO_AF)
{
    uint32_t temp = 0x00;
    uint32_t temp_2 = 0x00;

    /* Check the parameters */
```

(continues on next page)

(continued from previous page)

```

assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
assert_param(IS_GPIO_AF(GPIO_AF));

temp = ((uint32_t)(GPIO_AF) <<
           ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
GPIOx->AFR[GPIO_PinSource >> 0x03] &= ~((uint32_t)0xF <<
           ((uint32_t)((uint32_t)GPIO_PinSource & (uint32_t)0x07) * 4));
temp_2 = GPIOx->AFR[GPIO_PinSource >> 0x03] | temp;
GPIOx->AFR[GPIO_PinSource >> 0x03] = temp_2;
}

```

```

procedure Configure_Alternate_Function
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin;
   AF   : GPIO_Alternate_Function_Code)
is
begin
  Port.AFR (Pin) := AF;
end Configure_Alternate_Function;

```

Which one is correct? Both. But clearly, the Ada version is far simpler, so much so that it is immediately obvious that it is correct. Not so for the coding approach used in the C version, comparatively speaking. It is true that the Ada version required a couple more type declarations, but those make the procedure body far simpler. That resulting simplicity is a reflection of the balance between data structures and executable statements that we should always try to achieve. Ada just makes that easier to achieve than in some other languages.

Of course, the underlying hardware likely has no machine-supported 4-bit unsigned type so larger hardware numeric types are used in the generated code. Hence there are shifts and masking being done in the Ada version as well, but they do not appear in the source code. The developer has let the compiler do that work. An additional benefit of this approach is that the compiler will change the shifting and masking code for us if we change the explicit type declarations.

Why is simplicity so important? Simplicity directly increases understandability, which directly affects correctness and maintainability, which greatly affects the economic cost of the software. In large, long-lived projects, maintenance is by far the largest economic cost driver. In high-integrity applications, correctness is essential. Therefore, doing anything reasonable to keep the code as simple as possible is usually worth the effort. In some projects the non-functional requirements, especially performance, can dictate less simple code, but that won't apply to all of the code. Where possible, simplicity rules.

One more point about the GPIO ports. There are as many of these ports as the Arm microcontroller vendor decides to implement. And as we said, they are memory-mapped, at addresses specified by the vendor. If the memory used by all the ports is contiguous, we can conveniently use an array of the GPIO_Port record type to represent all the ports implemented. We would just set the array object's address at the address specified for the first port object in memory. Then, normal array indexing will provide access to any given port in the memory-mapped hardware.

This array approach requires each array component — the GPIO_Port record type — to be the right size so that all the array components start on addresses corresponding to the start of the next port in hardware.

That starting address correspondence for the array components is obtained automatically as long as the record type includes all the memory used by any individual device. In that case the next array component will indeed start at an address matching the next device in hardware. Note that this assumes the first array component matches the address of the first hardware device in memory. The first array component is at the same address as the

whole array object itself (a fact that is guaranteed by the language), so the array address must be set to whatever the vendor documentation specified for the first port.

However, in some cases the vendor will leave gaps of unused memory for complicated memory-mapped objects like these ports. They do so for the sake of future expansion of the implementation, e.g., to add new features or capacity. The gaps are thus between consecutive hardware devices.

These gaps are presumably (hopefully!) included in the memory layout documented for the device, but it won't be highlighted particularly. You should check, therefore, that the documented starting addresses of the second and subsequent array components are what you will get with a simple array object having components of that record type.

For example, the datasheet for the STM32F407 Arm implementation indicates that the GPIO ports start at address 16#4002_0000#. That's where GPIO_A begins. The next port, GPIO_B, starts at address 16#4002_0400#, or a byte offset of 1024 in decimal. In the STM32F4 Reference Manual, however, the GPIO port register layout indicates a size for any one port that is much less than 1024 bytes. As you saw earlier in the corresponding record type declaration, on the STM32F4 each port only requires 40 (decimal) bytes. Hence there's a gap of unused memory between the ports, including after the last port, of 984 bytes (7872 bits).

To represent the gap, an "extra", unused record component was added, with the necessary location and size specified within the record type, so that the unused memory is included in the representation. As a result, each array component will start at the right address (again, as long as the first one does). Telling the compiler, and future maintainers, that this extra component is not meant to be referenced by the software would not hurt. You can use the pragma or aspect Unreferenced for that purpose. Here's the code again, for convenience:

```
type GPIO_Port is limited record
    MODER      : Pin_Modes_Register;
    OTYPER     : Output_Types_Register;
    Reserved_1 : Half_Word;
    OSPEEDR   : Output_Speeds_Register;
    PUPDR     : Resistors_Register;
    IDR        : Half_Word;          -- input data register
    Reserved_2 : Half_Word;
    ODR        : Half_Word;          -- output data register
    Reserved_3 : Half_Word;
    BSRR_Set   : Half_Word;         -- bit set register
    BSRR_Reset : Half_Word;         -- bit reset register
    LCKR       : Word with Atomic;
    AFR        : Alternate_Function_Fields;
    Unused     : Unaccessed_Gap with Unreferenced;
end record with
Size => 16#400# * 8;

for GPIO_Port use record
    MODER      at 0 range 0 .. 31;
    OTYPER     at 4 range 0 .. 15;
    Reserved_1 at 6 range 0 .. 15;
    OSPEEDR   at 8 range 0 .. 31;
    PUPDR     at 12 range 0 .. 31;
    IDR        at 16 range 0 .. 15;
    Reserved_2 at 18 range 0 .. 15;
    ODR        at 20 range 0 .. 15;
    Reserved_3 at 22 range 0 .. 15;
    BSRR_Set   at 24 range 0 .. 15;
    BSRR_Reset at 26 range 0 .. 15;
    LCKR       at 28 range 0 .. 31;
    AFR        at 32 range 0 .. 63;
    Unused     at 40 range 0 .. 7871;
end record;
```

The type for the gap, `Unaccessed_Gap`, must represent 984 bytes so we declared an array like so:

```
Gap_Size : constant := 984; -- bytes
-- There is a gap of unused, reserved memory after the end of the
-- bytes used by any given memory-mapped GPIO port. The size of the
-- gap is indicated in the STM32F405xx etc. Reference Manual, RM 0090.
-- Specifically, Table 1 shows the starting and ending addresses mapped
-- to the GPIO ports, for an allocated size of 16#400#, or 1024 (decimal)
-- bytes per port. However, in the same document, the register map for
-- these ports shows only 40 bytes currently in use. Presumably this gap is
-- for future expansion when additional functionality or capacity is added,
-- such as more pins per port.

type Unaccessed_Gap is array (1 .. Gap_Size) of Unsigned_8 with
  Component_Size => Unsigned_8'Size,
  Size           => Gap_Size * Unsigned_8'Size;
-- This type is used to represent the necessary gaps between GPIO
-- ports in memory. We explicitly allocate a record component of
-- this type at the end of the record type for that purpose.
```

We also set the size of the entire record type to `16#400#` bytes since that is the total of the required bytes plus the gap, as per the documentation. As such, this is a "confirming" size clause because the reserved gap component increases the required size to that value (which is the point). We don't really need to do both, i.e., declare the reserved gap component and also set the record type size to the larger value. We could have done either one alone. One could argue that setting the size alone would have been simpler, in that it would obviate the type declaration and corresponding record component declaration. Being doubly explicit seemed a good idea at the time.

33.3 Dynamic Address Conversion

In the overlay example there were two distinct Ada objects, of two different types, sharing one (starting) address. The overlay provides two views of the memory at that address because there are two types involved. In this idiom the address is known when the code is written, either because it is a literal value specified in some hardware spec, or it is simply the address of the other object (in which case the actual address value is neither known nor relevant).

When there are several views required, declaring multiple overlaid variables at the same address absolutely can work, but can be less convenient than an alternative idiom. The alternative is to convert an address value to a value of an access type. Dereferencing the resulting access value provides a view of the memory corresponding to the designated type, starting at the converted address value.

For example, perhaps a networking component is given a buffer — an array of bytes — representing a received message. A subprogram is called with the buffer as a parameter, or the parameter can be the address of the buffer. If the subprogram must interpret this array via different views, this alternative approach works well. We could have an access type designating a message preamble, for example, and convert the first byte's address into such an access value. Dereferencing the conversion gives the preamble value. Likewise, the subprogram might need to compute a checksum over some of the bytes, so a different view, one of an array of a certain set size, could be used. Again, we could do that with overlaid objects but the alternative can be more convenient.

Here's a simple concrete example to illustrate the approach. Suppose we want to have a utility to swap the two bytes at any arbitrary address. Here's the declaration:

```
procedure Swap2 (Location : System.Address);
```

Callers pass the address of an object intended to have its (first) two bytes swapped:

```
Swap2 (Z'Address);
```

In the call, Z is of type Interfaces.Integer_16, for example, or Unsigned_16, or even something bigger as long as you only care about swapping the first two bytes.

The incomplete implementation using the conversion idiom could be like so:

```
procedure Swap2 (Location : System.Address) is
    X : Word renames To_Pointer (Location).all;
begin
    X := Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;
```

The declaration of X is the pertinent part.

In the declaration, X is of type Word, a type (not yet shown) derived from Interfaces.Unsigned_16. Hence X can have the inherited shift and logical **or** operations applied.

The To_Pointer (Location) part of the declaration is a function call. The function returns the conversion of the incoming address value in Location into an access value designating Word values. We'll explain how to do that momentarily. The **.all** explicitly dereferences the access value resulting from the function call.

Finally, X renames the Word designated by the converted access value. The benefit of the renaming, in addition to the simpler name, is that the function is only called once, and the access value deference is only evaluated once.

Now for the rest of the implementation not shown earlier.

```
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;
```

System.Address_To_Access_Conversions is a language-defined generic package that provides just two functions: one to convert an address value to an access type, and one to convert in the opposite direction:

```
generic
    type Object (<>) is limited private;
package System.Address_To_Access_Conversions is

    type Object_Pointer is access all Object;

    function To_Pointer (Value : Address) return Object_Pointer;
    function To_Address (Value : Object_Pointer) return Address;

    pragma Convention (Intrinsic, To_Pointer);
    pragma Convention (Intrinsic, To_Address);

end System.Address_To_Access_Conversions;
```

Object is the generic formal type parameter, i.e., the type we want our converted addresses to designate via the type Object_Pointer. In the byte-swapping example, the type Word was passed to Object in the instantiation.

The access type used by the functions is Object_Pointer, declared along with the functions. Object_Pointer designates values of the type used for the generic actual parameter, in this case Word.

Note the pragma Convention applied to each function, indicating that there is no actual function call involved; the compiler emits the code directly, if any code is actually required. Otherwise the compiler just treats the incoming **Address** bits as a value of type Object_Pointer.

The instantiation specifies type Word as the generic actual type parameter, so now we have a set of functions for that type, in particular To_Pointer.

Let's look at the code again, this time with the additional declarations:

```
type Word is new Interfaces.Unsigned_16;

package Word_Ops is new System.Address_To_Access_Conversions (Word);
use Word_Ops;

procedure Swap2 (Location : System.Address) is
    X : Word renames To_Pointer(Location).all;
begin
    X := Shift_Left (X, 8) or Shift_Right (X, 8);
end Swap2;
```

Word_Ops is the generic instance, followed immediately by a **use** clause so that we can refer to the visible content of the package instance conveniently.

In the renaming expression, To_Pointer (Location) converts the incoming address in Location to a pointer designating the Word at that address. The **.all** dereferences the resulting access value to get the designated Word value. Hence X refers to that two-byte value in memory.

We could almost certainly achieve the same affect by replacing the call to the function in To_Pointer with a call to an instance of Ada.Unchecked_Conversion. The conversion would still be between an access type and a value of type System.**Address**, but the access type would require declaration by the user. In both cases there would be an instantiation of a language-defined facility, so there's not much saving in lines of source code, other than the access type declaration. Because System.Address_To_Access_Conversions is explicitly intended for this purpose, good style suggests its use in preference to unchecked conversion, but both approaches are common in production code.

In either case, the conversion is not required to work, although in practice it will, most of the time. Representing an access value as an address value is quite common because it matches the typical underlying hardware's memory model. But even so, a single address is not necessarily sufficient to represent an access value for any given designated type. In that case problems arise, and they are difficult to debug.

For example, in GNAT, access values designating values of unconstrained array types, such as **String**, are represented as two addresses, known as "fat pointers". One address points to the bounds for the specific array object, since they can vary. The other address designates the characters. Therefore, conversions of a single address to an access value requiring fat pointers will not work using unchecked conversions. (There is a way, however, to tell GNAT to use a single address value, but it is an explicit step in the code. Once done, though, unchecked conversions would then work correctly.)

You can alternatively use generic package System.Address_To_Access_Conversions. That generic is defined for the purpose of converting addresses to access values, and vice versa. But note that the implementation of the generic's routines must account for the representation their compiler uses for unbounded types like **String**.

33.4 Address Arithmetic

Part of "letting the compiler do the work for you" is not doing address arithmetic in the source code if you can avoid it. Instead, for instance, use the normal "dot notation" to reference components, and let the compiler compute the offsets to those components. The approach to implementing procedure `Configure_Alternate_Function` for a `GPIO_Port` is a good example.

That said, sometimes address arithmetic is the most direct expression of what you're trying to implement. For example, when implementing your own memory allocator, you'll need to do address arithmetic.

Earlier in this section we mentioned the package `System.Storage_Elements`, for the sake of the function that converts integer values to address values. The package also defines functions that provide address arithmetic. These functions work in terms of type `System.Address` and the package-defined type `Storage_Offset`. The type `Storage_Offset` is an integer type with an implementation-defined range. As a result you can have positive and negative offsets, as needed. Addition and subtraction of offsets to/from addresses is supported, as well as the `mod` operator.

Combined with package `System` (for type `System.Address`), the functions and types in this package provide the kinds of address arithmetic other languages provide. Nevertheless, you should prefer having the compiler do these computations for you, if possible.

Here's an example illustrating the facilities. The procedure defines an array of record values, then traverses the array, printing the array components as it goes. (This is not the way to really implement such code. It's just an illustration for address arithmetic.)

```

with Ada.Text_Io;           use Ada.Text_Io;
with System.Storage_Elements; use System.Storage_Elements;
with System.Address_To_Access_Conversions;

procedure Demo_Address_Arithmetic is

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

    R_Size : constant Storage_Offset := R'Object_Size / System.Storage_Unit;
    Objects : aliased array (1 .. 10) of aliased R;      -- arbitrary bounds
    Objects_Base : constant System.Address := Objects'Address;
    Offset : Storage_Offset;

    -- display the object of type R at the address specified by Location
    procedure Display_R (Location : in System.Address) is

        package R_Pointers is new System.Address_To_Access_Conversions (R);
        use R_Pointers;

        Value : R renames To_Pointer (Location).all;
        -- The above converts the address to a pointer designating an R value
        -- and dereferences it, using the name Value to refer to the
        -- dereferenced R value.

    begin
        Put (Integer'Image (Value.X));
        Put ",";
        Put (Integer'Image (Value.Y));

```

(continues on next page)

(continued from previous page)

```
New_Line;
end Display_R;

begin
    Objects := ((0,0), (1,1), (2,2), (3,3), (4,4),
                (5,5), (6,6), (7,7), (8,8), (9,9));

    Offset := 0;

    -- walk the array of R objects, displaying each one individually by
    -- adding the offset to the base address of the array
    for K in Objects'Range loop
        Display_R (Objects_Base + Offset);
        Offset := Offset + R_Size;
    end loop;
end Demo_Address_Arithmetic;
```

Seriously, this is just for the purpose of illustration. It would be much better to just index into the array directly.

CHAPTER
THIRTYFOUR

GENERAL-PURPOSE CODE GENERATORS

In *another chapter* (page 435), we mentioned that the best way to get a specific set of machine instructions emitted from the compiler is to write them ourselves, in the Ada source code, using machine-code insertions (MCI). The rationale was that the code generator will make reasonable assumptions, including the assumption that performance is of uppermost importance, but that these assumptions can conflict with device requirements.

For example, the code generator might not issue the specific sequence of machine code instructions required by the hardware. The GPIO pin "lock" sequence in that referenced chapter is a good example. Similarly, the optimizer might remove what would otherwise be "redundant" read/writes to a memory-mapped variable.

The code generator might issue instructions to read a small field in a memory-mapped record object using byte-sized accesses, when instead the device requires whole-word or half-word access instructions.

The code generator might decide to load a variable from memory into a register, accessing the register when the value is required. Typically that approach will yield far better performance than going to memory every time the value is read or updated. But suppose the variable is for a memory-mapped device? In that case we really need the generated code to go to memory every time.

As you can see, there are times when we cannot let the code generator make the usual assumptions. Therefore, Ada provides aspects and pragmas that developers can use to inform the compiler of facts that affect code generation in this regard.

These facilities are defined in the Systems Programming Annex, C.6, specifically. The title of that sub-clause is "Shared Variables" because the objects (memory) can be shared between tasks as well as between hardware devices and the host computer. We ignore the context of variables shared between tasks, focusing instead of shared memory-mapped devices, as this course is about embedded systems.

When describing these facilities we will use aspects, but remember that the corresponding pragmas are defined as well, except for one. (We'll mention it later.) For the other aspects, the pragmas existed first and, although obsolescent, remain part of the language and supported. There's no need to change your existing source code using the pragmas to use the aspects instead, unless you need to change it for some other reason.

As this is an introduction, we will not go into absolutely all the details, but will instead give a sense of what the language provides, and why.

34.1 Aspect Independent

To interface with a memory-mapped device, there will be an Ada object of an appropriate type that is mapped to one or more bytes of memory. The software interacts with the device by reading and/or writing to the memory locations mapped to the device, using the operations defined by the type in terms of normal Ada semantics.

Some memory-mapped devices can be directly represented by a single scalar value, usually of some signed or unsigned numeric type. More sophisticated devices almost always involve several distinct input and output fields. Therefore, representation in the software as a record object is very common. Ada record types have such extensive and flexible support for controlling their representation, down to the individual bit level, that using a record type makes sense. (And as mentioned, using normal record component access via the "dot notation" offloads to the compiler the address arithmetic needed to access individual memory locations mapped to the device.) And of course the components of the mapped record type can themselves be of scalar and composite types too, so an extensive descriptive capability exists with Ada.

Let's say that one of these record components is smaller than the size of the smallest addressable memory unit on the machine, which is to say, smaller than the machine instructions can read/write memory individually. A Boolean record component is a good example, and very common. The machine cannot usually read/write single bits in memory, so the generated code will almost certainly read or write a byte to get the enclosed single-bit Boolean component. It might use a larger sized access too, a half-word or word. Then the generated code masks off the bits that are not of interest and does some shifts to get the desired component.

Reading and writing the bytes surrounding the component accessed in the source code can cause a problem. In particular, some devices react to being read or written by doing something physical in the hardware. That's the device designer's intent for the software. But we don't want that to happen accidentally due to surrounding bytes being accessed.

Therefore, to prevent these "extra" bytes from being accessed, we need a way to tell the compiler that we need the read or write accesses for the given object to be independent of the surrounding memory. If the compiler cannot do so, we'll get an error and the compilation will fail. That beats debugging, every time.

Therefore, the aspect `Independent` specifies that the code generated by the compiler must be able to load and store the memory for the specified object without also accessing surrounding memory. More completely, it declares that a type, object, or component must be independently addressable by the hardware. If applied to a type, it applies to all objects of the type.

Likewise, aspect `Independent_Components` declares that the individual components of an array or record type must be independently addressable.

With either aspect the compiler will reject the declaration if independent access is not possible for the type/object in question.

For example, if we try to mark each Boolean component of a record type as `Independent` we can do so, either individually or via `Independent_Components`, but doing so will require that each component is a byte in size (or whatever the smallest addressable unit happens to be on this machine). We cannot make each Boolean component occupy one bit within a given byte if we want them to be independently accessed.

```
package P is
    type R is record
        B0 : Boolean;
        B1 : Boolean;
        B2 : Boolean;
```

(continues on next page)

(continued from previous page)

```

B3 : Boolean;
B4 : Boolean;
B5 : Boolean;
end record with
Size => 8,
Independent_Components;

for R use record
  B0 at 0 range 0 .. 0;
  B1 at 0 range 1 .. 1;
  B2 at 0 range 2 .. 2;
  B3 at 0 range 3 .. 3;
  B4 at 0 range 4 .. 4;
  B5 at 0 range 5 .. 5;
end record;

end P;

```

For a typical target machine the compiler will reject that code, complaining that the Size for R' must be at least 48 bits, i.e., 8 bits per component. That's because the smallest quantity this machine can independently address is an 8-bit byte.

But if we don't really need the individual bits to be independently accessed — and let's hope no hardware designer would define such a device — then we have more flexibility. We could, for example, require that objects of the entire record type be independently accessible:

```

package Q is

  type R is record
    B0 : Boolean;
    B1 : Boolean;
    B2 : Boolean;
    B3 : Boolean;
    B4 : Boolean;
    B5 : Boolean;
  end record with
  Size => 8,
  Independent;

  for R use record
    B0 at 0 range 0 .. 0;
    B1 at 0 range 1 .. 1;
    B2 at 0 range 2 .. 2;
    B3 at 0 range 3 .. 3;
    B4 at 0 range 4 .. 4;
    B5 at 0 range 5 .. 5;
  end record;

end Q;

```

This the compiler should accept, assuming a machine that can access bytes in memory individually, without having to read some number of other bytes.

But for another twist, suppose we need one of the components to be aliased, so that we can construct access values designating it via the **Access** attribute? For example, given the record type R above, and some object Foo of that type, suppose we want to say Foo.B0'Access? We'd need to mark the component as **aliased**:

```

package QQ is

  type R is record

```

(continues on next page)

(continued from previous page)

```

B0 : aliased Boolean;
B1 : Boolean;
B2 : Boolean;
B3 : Boolean;
B4 : Boolean;
B5 : Boolean;
end record with
  Size => 8,
  Independent;

for R use record
  B0 at 0 range 0 .. 0;
  B1 at 0 range 1 .. 1;
  B2 at 0 range 2 .. 2;
  B3 at 0 range 3 .. 3;
  B4 at 0 range 4 .. 4;
  B5 at 0 range 5 .. 5;
end record;

end QQ;

```

The compiler will once again reject the code, complaining that the size of B0 must be a multiple of a `Storage_Unit`, in other words, the size of something independently accessible in memory on this machine.

Why? The issue here is that aliased objects, including components of composite types, must be represented in such a way that creating the designating access ("pointer") value is possible. The component B0, if allocated only one bit, would not allow an access value to be created due to the usual machine accessibility limitation we've been discussing.

Similarly, a record component that is of some by-reference type, such as any tagged type, introduces the same issues as an aliased component. That's because the underlying implementation of by-reference parameter passing is much like a '`Access` attribute reference.

As important as the effect of this aspect is, you probably won't see it specified. There are other aspects that are more typically required. However, the semantics of `Independent` are part of the semantics of some of these other aspects. Applying them applies `Independent` too, in effect. So even though you don't typically apply it directly, you need to understand the independent access semantics. We discuss these other, more commonly applied aspects next.

These representation aspects may be specified for an object declaration, a component declaration, a full type declaration, or a generic formal (complete) type declaration. If any of these aspects are specified True for a type, then the corresponding aspect is True for all objects of the type.

34.2 Aspect Volatile

Earlier we said that the compiler (specifically the optimizer) might decide to load a variable from memory into a register, accessing the register when the value is required or updated. Similarly, the compiler might reorder instructions, and remove instructions corresponding to redundant assignments in the source code. Ordinarily we'd want those optimizations, but in the context of embedded memory-mapped devices they can be problematic.

The hardware might indeed require the source code to read or write to the device in a way that the optimizer would consider redundant, and in order to interact with the device we need every read and write to go to the actual memory for the mapped device, rather than a register. As developers we have knowledge about the context that the compiler lacks.

The compiler is aware of the fact that the Ada object is memory-mapped because of the address clause placing the object at a specific address. But the compiler does not know we are interacting with an external hardware device. Perhaps, instead, the object is mapped to a specific location because some software written in another language expects to access it there. In that case redundant reads or writes of the same object really would be redundant. The fact that we are interacting with a hardware device makes a difference.

In terms of the language rules, we need reading from, and writing to, such devices to be part of what the language refers to as the "external effects" of the software. These effects are what the code must actually produce. Anything else — the internal effects — could be removed by the optimizer.

For example, suppose you have a program that writes a value to some variable and also writes the string literal "42" to a file. That's is absolutely all that the program contains.

```
with Ada.Text_IO;  use Ada.Text_IO;

procedure Demo is
    Output : File_Type;
    Silly : Integer;
begin
    Silly := 0;
    Create (Output, Out_File, "output.txt");
    Put (Output, "42");
    Close (Output);
end Demo;
```

The value of the variable `Silly` is not used in any way so there is no point in even declaring the variable, much less generating code to implement the assignment. The update to the variable has only an internal effect. With warnings enabled we'll receive notice from the compiler, but they're just warnings.

However, writing to the file is an external effect because the file persists beyond the end of the program's execution. The optimizer (when enabled) would be free to remove any access to the variable `Silly`, but not the write to the file.

We can make the compiler recognize that a software object is part of an external effect by applying the aspect `Volatile`. (Aspect `Atomic` is pertinent too. More in a moment.) As a result, the compiler will generate memory load or store instructions for every read or update to the object that occurs in the source code. Furthermore, it cannot generate any additional loads or stores to that variable, and it cannot reorder loads or stores from their order in the source code. "What You See Is What You Get" in other words.

```
with Ada.Text_IO;  use Ada.Text_IO;

procedure Demo is
    Output : File_Type;
    Silly : Integer with Volatile;
begin
    Silly := 0;
    Create (Output, Out_File, "output.txt");
    Put (Output, "42");
    Close (Output);
end Demo;
```

If we compile the above, we won't get the warning we got earlier because the compiler is now required to generate the assignment for `Silly`.

The variable `Silly` is not even a memory-mapped object, but remember that we said these aspects are important to the tasking context too, for shared variables. We're ignoring that context in this course.

There is another reason to mark a variable as `Volatile`. Sometimes you want to have

exactly the load and store instructions generated that match those of the Ada code, even though the volatile object is not a memory-mapped object. For example, *elsewhere* (page 435) we said that the best way to achieve exact assembly instruction sequences is the use of machine-code inserts (MCIs). That's true, but for the moment let's say we want to write it in Ada without the MCIs. Our earlier example was the memory-mapped GPIO ports on Arm microcontrollers produced by ST Microelectronics. Specifically, these ports have a "lock" per GPIO pin that allows the developer to configure the pin and then lock it so that no other configuration can accidentally change the configuration of that pin. Doing so requires an exact sequence of loads and stores. If we wrote this in Ada it would look like this:

```
procedure Lock
  (Port : in out GPIO_Port;
   Pin  : GPIO_Pin)
is
  Temp : Word with Volatile;
begin
  -- set the lock control bit and the pin
  -- bit, clear the others
  Temp := LCCK or Pin'Enum_Rep;

  -- write the lock and pin bits
  Port.LCKR := Temp;

  -- clear the lock bit in the upper half
  Port.LCKR := Pin'Enum_Rep;

  -- write the lock bit again
  Port.LCKR := Temp;

  -- read the lock bit
  Temp := Port.LCKR;

  -- read the lock bit again
  Temp := Port.LCKR;
end Lock;
```

Temp is marked volatile for the sake of getting exactly the load and stores that we express in the source code, corresponding to the hardware locking protocol. It's true that Port is a memory-mapped object, so it too would be volatile, but we also need Temp to be volatile.

This high-level coding approach will work, and is simple enough that MCIs might not be needed. However, what really argues against it is that the correct sequence of emitted code requires the optimizer to remove all the other cruft that the code generator would otherwise include. (The gcc code generator used by the GNAT compiler generates initially poor code, by design, relying on the optimizer to clean it up.) In other words, we've told the optimizer not to change or add loads and stores for Temp, but without the optimizer enabled the code generator generates other code that gets in the way. That's OK in itself, as far as procedure Lock is concerned, but if the optimizer is sufficiently enabled we cannot debug the rest of the code. Using MCIs avoids these issues. The point, though, is that not all volatile objects are memory mapped.

So far we've been illustrating volatility with scalar objects, such as Lock.Temp above. What about objects of array and record types? (There are other "composite" types in Ada but they are not pertinent here.)

When aspect Volatile is applied to a record type or an object of such a type, all the record components are automatically volatile too.

For an array type (but not a record type), a related aspect Volatile_Components declares that the components of the array type — but not the array type itself — are volatile. However, if the Volatile aspect is specified, then the Volatile_Components aspect is automatically applied too, and vice versa. Thus components of array types are covered auto-

matically.

If an object (of an array type or record type) is marked volatile then so are all of its sub-components, even if the type itself is not marked volatile.

Therefore aspects Volatile and Volatile_Components are nearly equivalent. In fact, Volatile_Components is superfluous. The language provides the Volatile_Components aspect only to give symmetry with the Atomic_Components and Independent_Components aspects. You can simply apply Volatile and be done with it.

Finally, note that applying aspect Volatile does not implicitly apply Independent, although you can specify it explicitly if need be.

34.3 Aspect Atomic

Consider the GPIO pin configuration lock we've mentioned a few times now, that freezes the configuration of a given pin on a given GPIO port. The register, named LCKR for "lock register", occupies 32-bits, but only uses 17 total bits (currently). The low-order 16 bits, [0:15], represent the 16 GPIO pins on the given port. Bit #16 is the lock bit. That bit is the first bit in the upper half of the entire word. To freeze the configuration of a given pin in [0:15], the lock bit must be set at the same time as the bit to be frozen. In other words, the lower half and the upper half of the 32-bit word representing the register must be written together, at the same time. That way, accidental (un)freezing is unlikely to occur, because the most efficient, hence typical way for the generated code to access individual bits is for the compiler to load or store just the single byte that contains the bit or bits in question.

This indivisibility effect can be specified via aspect Atomic. As a result, all reads and updates of such an object as a whole are indivisible. In practice that means that the entire object is accessed with one load or store instruction. For a 16-bit object, all 16-bits are loaded and stored at once. For a 32-bit object, all 32-bits at once, and so on. The upper limit is the size of the largest machine scalar that the processor can manipulate with one instruction, as defined by the target processor. The typical lower bound is 8, for a byte-addressable machine.

Therefore, within the record type representing a GPIO port, we include the lock register component and apply the aspect Atomic:

```
type GPIO_Port is limited record
  ...
  LCKR : UInt32 with Atomic;
  ...
end record with
...
Size => 16#400# * 8;
```

Hence loads and stores to the LCKR component will be done atomically, otherwise the compiler will let us know that it is impossible. That's all we need to do for the lock register to be read and updated atomically.

You should understand that only accesses to the whole, entire object are atomic. In the case of the lock register, the entire object is a record component, but that causes no problems here.

There is, however, something we must keep in mind when manipulating the values of atomic objects. For the lock register we're using a scalar type to represent the register, an unsigned 32-bit integer. There are no sub-components because scalar types don't have components, by definition. We simply use the bit-level operations to set and clear the individual bits. But we cannot set the bits — the lock bit and the bit for the I/O pin to freeze — one at a time because the locking protocol requires all the bits to be written at the same time, and only the entire 32-bit load and stores are atomic. Likewise, if instead of a scalar we used

a record type or an array type to represent the bits in the lock register, we could not write individual record or array components one at a time, for the same reason we could not write individual bits using the unsigned scalar. The `Atomic` aspect only applies to loads and stores of the entire register.

Therefore, to update or read individual parts of an atomic object we must use a coding idiom in which we explicitly read or write the entire object to get to the parts. For example, to read an individual record component, we'd first read the entire record object into a temporary variable, and then access the component of that temporary variable. Likewise, to update one or more individual components, we'd first read the record object into a temporary variable, update the component or components within that temporary, and then write the temporary back to the mapped device object. This is known as the "read-modify-write" idiom. You'll see this idiom often, regardless of the programming language, because the hardware requirement is not unusual. Fortunately Ada defines another aspect that makes the compiler do this for us. We'll describe it in the next section.

Finally, there are issues to consider regarding the other aspects described in this section.

If you think about atomic behavior in the context of machine instructions, loading and storing from/to memory atomically can only be performed for quantities that are independently addressable. Consequently, all atomic objects are considered to be specified as independently addressable too. Aspect specifications and representation items cannot change that fact. You can expect the compiler to reject any aspect or representation choice that would prevent this from being true.

Likewise, atomic accesses only make sense on actual memory locations, not registers. Therefore all atomic objects are volatile objects too, automatically.

However, unlike volatile objects, the components of an atomic object are not automatically atomic themselves. You'd have to mark these types or objects explicitly, using aspect `Atomic_Components`. Unlike `Volatile_Components`, aspect `Atomic_Components` is thus useful.

As is usual with Ada programming, you can rely on the compiler to inform you of problems. The compiler will reject an attempt to specify `Atomic` or `Atomic_Components` for an object or type if the implementation cannot support the indivisible and independent reads and updates required.

34.4 Aspect Full_Access_Only

Many devices have single-bit flags in the hardware that are not allocated to distinct bytes. They're packed into bytes and words shared with other flags. It isn't just individual bits either. Multi-bit fields that are smaller than a byte, e.g., two 4-bit quantities packed into a byte, are common. We saw that with the GPIO alternate functions codes earlier.

Ordinarily in Ada we represent such composite hardware interfaces using a record type. (Sometimes an array type makes more sense. That doesn't change anything here.) Compared to using bit-patterns, and the resulting bit shifting and masking in the source code, a record type representation and the resulting "dot notation" for accessing components is far more readable. It is also more robust because the compiler does all the work of retrieving these individual bits and bit-fields for us, doing any shifting and masking required in the generated code. The loads and stores are done by the compiler in whatever manner the compiler thinks most efficient.

When the hardware device requires atomic accesses to the memory mapped to such flags, we cannot let the compiler generate whatever width load and store accesses it thinks best. If full-word access is required, for example, then only loads and stores for full words can work. Yet aspect `Atomic` only guarantees that the entire object, in this case the record object, is loaded and stored indivisibly, via one instruction. The aspect doesn't apply to reads and updates to individual record components.

In the section on Atomic above, we mentioned that proper access to individual components of atomic types/objects can be achieved by a "read-modify-write" idiom. In this idiom, to read a component you first read into a temporary the entire enclosing atomic object. Then you read the individual component from that temporary variable. Likewise, to update an individual component, you start with the same approach but then update the component(s) within the temporary, then store the entire temporary back into the mapped atomic object. Applying aspect Atomic to the enclosing object ensures that reading and writing the temporary will be atomic, as required.

Using bit masks and bit patterns to access logical components as an alternative to a record type doesn't change the requirement for the idiom.

Consider the STM32F4 DMA device. The device contains a 32-bit stream configuration register that requires 32-bit reads and writes. We can map that register to an Ada record type like so:

```
type Stream_Config_Register is record
    ...
    Direction      : DMA_Data_Transfer_Direction;
    P_Flow_Controller : Boolean;
    TCI_Enabled    : Boolean; -- transfer complete
    HTI_Enabled    : Boolean; -- half-transfer complete
    TEI_Enabled    : Boolean; -- transfer error
    DMEI_Enabled   : Boolean; -- direct mode error
    Stream_Enabled : Boolean;
end record
with Atomic, Size => 32;
```

The "confirming" size clause ensures we have declared the type correctly such that it will fit into 32-bits. There will also be a record representation clause to ensure the record components are located internally as required by the hardware. We don't show that part.

The aspect Atomic is applied to the entire record type, ensuring that the memory mapped to the hardware register is loaded and stored only as 32-bit quantities. In this example it isn't that we want the loads and stores to be indivisible. Rather, we want the generated machine instructions that load and store the object to use 32-bit word instructions, even if we are only reading or updating a component of the object. That's what the hardware requires for all accesses.

Next we'd use that type declaration to declare one of the components of an enclosing record type representing one entire DMA "stream":

```
type DMA_Stream is record
    CR      : Stream_Config_Register;
    NDTR   : Word; -- upper half must remain at reset value
    PAR    : Address; -- peripheral address register
    M0AR   : Address; -- memory 0 address register
    M1AR   : Address; -- memory 1 address register
    FCR    : FIFO_Control_Register;
end record
with Volatile, Size => 192; -- 24 bytes
```

Hence any individual DMA stream record object has a component named CR that represents the corresponding configuration register.

The DMA controllers have multiple streams per unit so we'd declare an array of DMA_Stream components. This array would then be part of another record type representing a DMA controller. Objects of the DMA_Controller type would be mapped to memory, thus mapping the stream configuration registers to memory.

Now, given all that, suppose we want to enable a stream on a given DMA controller. Using the read-modify-write idiom we would do it like so:

```

procedure Enable
  (Unit    : in out DMA_Controller;
   Stream  : DMA_Stream_Selector)
is
  Temp : Stream_Config_Register;
  -- these registers require 32-bit accesses, hence the temporary
begin
  Temp := Unit.Streams (Stream).CR; -- read entire CR register
  Temp.Stream_Enabled := True;
  Unit.Streams (Stream).CR := Temp; -- write entire CR register
end Enable;

```

That works, and of course the procedural interface presented to clients hides the details, as it should.

To be fair, the bit-pattern approach can express the idiom concisely, as long as you're careful. Here's the C code to enable and disable a selected stream:

```

#define DMA_SxCR_EN ((uint32_t)0x00000001)

/* Enable the selected DMAy Streamx by setting EN bit */
DMAy_Streamx->CR |= DMA_SxCR_EN;

/* Disable the selected DMAy Streamx by clearing EN bit */
DMAy_Streamx->CR &= ~DMA_SxCR_EN;

```

The code reads and writes the entire CR register each time it is referenced so the requirement is met.

Nevertheless, the idiom is error-prone. We might forget to use it at all, or we might get it wrong in one of the very many places where we need to access individual components.

Fortunately, Ada provides a way to have the compiler implement the idiom for us, in the generated code. Aspect `Full_Access_Only` specifies that all reads of, or writes to, a component are performed by reading and/or writing all of the nearest enclosing full access object. Hence we add this aspect to the declaration of `Stream_Config_Register` like so:

```

type Stream_Config_Register is record
  -- ...
  Direction      : DMA_Data_Transfer_Direction;
  P_Flow_Controller : Boolean;
  TCI_Enabled     : Boolean; -- transfer complete interrupt
  HTI_Enabled     : Boolean; -- half-transfer complete
  TEI_Enabled     : Boolean; -- transfer error interrupt
  DMEI_Enabled    : Boolean; -- direct mode error interrupt
  Stream_Enabled  : Boolean;
end record
with Atomic, Full_Access_Only, Size => 32;

```

Everything else in the declaration remains unchanged.

Note that `Full_Access_Only` can only be applied to `Volatile` types or objects. `Atomic` types are automatically `Volatile` too, so either one is allowed. You'd need one of those aspects anyway because `Full_Access_Only` just specifies the accessing instruction requirements for the generated code when accessing components.

The big benefit comes in the source code accessing the components. Procedure `Enable` is now merely:

```

procedure Enable
  (Unit    : in out DMA_Controller;
   Stream  : DMA_Stream_Selector)
is

```

(continues on next page)

(continued from previous page)

```
begin
  Unit.Streams (Stream).CR.Stream_Enabled := True;
end Enable;
```

This code works because the compiler implements the read-modify-write idiom for us in the generated code.

The aspect `Full_Access_Only` is new in Ada 2022, and is based on an implementation-defined aspect that GNAT first defined named `Volatile_Full_Access`. You'll see that GNAT aspect throughout the Arm device drivers in the Ada Drivers Library, available here: https://github.com/AdaCore/Ada_Drivers_Library. Those drivers were the motivation for the GNAT aspect.

Unlike the other aspects above, there is no pragma corresponding to the aspect `Full_Access_Only` defined by Ada 2022. (There is such a pragma for the GNAT-specific version named `Volatile_Full_Access`, as well as an aspect.)

HANDLING INTERRUPTS

35.1 Background

Embedded systems developers offload functionality from the application processor onto external devices whenever possible. These external devices may be on the same "chip" as the central processor (e.g., within a System-on-Chip) or they may just be on the same board, but the point here is that they are not the processor executing the application. Offloading work to these other devices enables us to get more functionality implemented in a target platform that is usually very limited in resources. If the processor has to implement everything we might miss deadlines or perhaps not fit into the available code space. And, of course, some specialized functionality may simply require an external device, such as a sensor.

For a simple example, a motor encoder is a device attached to a motor shaft that can be used to count the number of full or partial rotations that the shaft has completed. When the shaft is rotating quickly, the application would need to interact with the encoder frequently to get an up-to-date count, representing a non-trivial load on the application processor. There are ways to reduce that load, which we discuss shortly, but by far the simplest and most efficient approach is to do it all in hardware: use a timer device driven directly by the encoder. The timer is connected to the encoder such that the encoder signals act like an external clock driving the timer's internal counter. All the application processor must do to get the encoder count is query the timer's counter. The timer is almost certainly memory-mapped, so querying the timer amounts to a memory access.

In some cases, we even offload communication with these external devices onto other external devices. For example, the I₂C⁴⁸ (Inter-Integrated Circuit) protocol is a popular two-wire serial protocol for communicating between low-level hardware devices. Individual bits of the data are sent by driving the data line high and low in time with the clock signal on the other line. The protocol has been around for a long time and many embedded devices use it to communicate. We could have the application drive the data line for each individual bit in the protocol. Known as "bit-banging," that would be a significant load on the processor when the overall traffic volume is non-trivial. Fortunately, there are dedicated devices — I₂C transceivers — that will implement the protocol for us. To send application data to another device using the I₂C protocol, we just give the transceiver the data and destination address. The rest is done in the transceiver hardware. Receiving data is of course also possible. I₂C transceivers are ubiquitous because the protocol is so common among device implementations. A USART⁴⁹ / UART⁵⁰ is a similar example.

Having offloaded some of the work, the application must have some way to interact with the device in order to know what is happening. Maybe the application has requested the external device perform some service — an analog-to-digital conversion, say — and must know when that function has completed. Maybe a communications device is receiving

⁴⁸ <https://en.wikipedia.org/wiki/I%C2%BC2>

⁴⁹ https://en.wikipedia.org/wiki/Universal_synchronous_and_asynchronous_receiver-transmitter

⁵⁰ https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

incoming data for the application to process. Or maybe that communications device has completed sending outgoing data and is ready for more to send.

Ultimately, interaction with the external device will be either synchronous or asynchronous, and has system-level design implications.

For synchronous interaction, the application periodically queries the device, typically a status flag or function on the device. Known as "polling," this approach is simple to implement but wastes cycles when the external device has not yet completed the request. After all, the point of offloading the work is to allow the application processor to execute other functionality. Polling negates that benefit. On the other hand, if the expected time to completion is extremely short, polling can be sufficiently efficient to make sense.

Usually, there's enough time involved so that polling is undesirable. The external environment takes time to respond and change state. Maybe a sensor has been designed to wait passively for something to happen in the external world, and only on the infrequent occurrence of that event should the application be notified. Perhaps a switch is to be toggled in certain circumstances, or an intruder detected. In this case, nothing happens for extended intervals.

As a consequence of all this, there's a very good chance that the internal processor should not poll these external devices.

Before we discuss the asynchronous alternative, there's another issue to consider. However the notification from the external device is implemented, a very quick response from the internal processor may be required. Think back to that serial port with a USART again. The USART is responsible for composing the arriving characters (or bytes) from their individual incoming bits on the receiving line. When all the bits for a single character have arrived, what happens next depends on the software design. In the simplest case, the internal processor copies the single character from the USART to an internal buffer and then goes back to doing something else while the next full character arrives in the USART. The response to the USART must be fairly quick because the next incoming character's bits are arriving. The internal processor must get the current character before it is overwritten by the next arriving character, otherwise we'll lose data. So we can say that the response to the notification from the external device must often be very quick.

Now, ideally in the USART case, we would further offload the work from the internal processor. Instead of having the processor copy each arriving character from the USART into an application buffer, we would have another external hardware device — a [direct memory access \(DMA\)](#)⁵¹ device — copy each arriving character from the USART to the buffer. A DMA device copies data from one location to another, in this case from the address of the USART's one-character memory-mapped register to the address of the application buffer in memory. The copy is performed by the DMA hardware so it is extremely fast and costs the main processor no cycles. But even with this approach, we need to notify the application that a complete message is ready for processing. We might need to do that quickly so that enough time remains for the application to process the message content prior to the arrival of the next message.

Therefore, the general requirement is for an external device to be able to asynchronously notify the internal processor, and for the notification to be implemented in such a way that the beginning of the response can be sufficiently and predictably quick.

Fortunately, computers already have such a mechanism: interrupts. The details vary considerably with the hardware architecture, but the overall idea is independent of the [ISA](#)⁵²: an external event can trigger a response from the processor by becoming "active." The current state of the application is temporarily stored, and then an interrupt response routine, known as an "interrupt handler" is executed. Upon completion of the handler, the original state of the application is restored and the application continues execution. The time between the interrupt becoming active and the start of the responding handler execution is known as the "interrupt latency."

⁵¹ https://en.wikipedia.org/wiki/Direct_memory_access

⁵² https://en.wikipedia.org/wiki/Instruction_set_architecture

Hardware interrupts typically have priorities assigned, depending on the hardware. These priorities are applied when multiple interrupts are triggered at the same time, to define the order in which the interrupts are presented and the handlers invoked. The canonical model is that only higher-priority interrupts can preempt handlers executing in response to interrupts with lower or equal priority.

Ada defines a model for hardware interrupts and interrupt handling that closely adheres to the conceptual model described above. If you have experience with interrupt handling, you will recognize them in the Ada model. One very important point to make about the Ada facilities is that they are highly portable, so they don't require extensive changes when moving to a new target computer. Part of that portability is due to the language-defined model.

Before we go into the Ada facility details, there's a final point. Sometimes we *do* want the application to wait for the external device. When would that be the case? To answer that, we need to introduce another term. The act of saving and restoring the state of the interrupted application software is known as "interrupt context switching." If the time for the device to complete the application request is approximately that of the context switching, the application might as well wait for the device after issuing the request.

Another reason to consider polling is that the architectural complexity of interrupt handling is greater than that of polling. If your system has some number of devices to control and polling them would be fast enough for the application to meet requirements, it is simpler to do so. But that will likely only work for a few devices, or at least a few that have short response time requirements.

The application code can wait for the device by simply entering a loop, exiting only when some external device status flag indicates completion of the function. The loop itself, in its simplest form, would contain only the test for exiting. As mentioned earlier, polling in a tight loop like this only makes sense for very fast device interactions. That's not the usual situation though, so polling should not be your default design assumption. Besides, active polling consumes power. On an embedded platform, conserving power is often important.

That loop polling the device will never exit if the device can fail to signal completion. Or maybe it might take too long in some odd case. If you don't want to be potentially stuck in the loop indefinitely, chewing up cycles and power, you can add an upper bound on the number of attempts, i.e., loop iterations. For example:

```
procedure Await_Data_Ready (This : in out Three_Axis_Gyroscope) is
  Max_Status_Attempts : constant := 10_000;
  -- This upper bound is arbitrary but must be sufficient for the
  -- slower gyro data rate options and higher clock rates. It need
  -- not be as small as possible, the point is not to hang forever.
begin
  Polling: for K in 1 .. Max_Status_Attempts loop
    if Data_Status (This).ZYX_Available then
      return;
    end if;
  end loop Polling;
  raise Gyro_Failure;
end Await_Data_Ready;
```

In the above, *Data_Status* is a function that returns a record object containing Boolean flags. The *if*-statement queries one of those flags. Thus the loop either detects the desired device status or raises an exception after the maximum number of attempts have been made. In this version, the maximum is a known upper bound so a local constant will suffice. The maximum could be passed as a parameter instead, or declared in a global "configuration" package containing such constants.

Presumably, the upper bound on the attempts is either specified by the device documentation or empirically determined. Sometimes, however, the documentation will instead specify a maximum possible response time, for instance 30 milliseconds. Any time beyond

that maximum indicates a device failure.

In the code above, the number of iterations indirectly defines the amount of elapsed time the caller waits. That time varies with the target's system clock and the generated instructions' required clock cycles, hence the approach is not portable. Alternatively, we can work in terms of actual time, which will be portable across all targets with a sufficiently precise clock.

You can use the facilities in package `Ada.Real_Time` to work with time values. That package defines a type `Time_Span` representing time intervals, useful for expressing relative values such as elapsed time. There is also type `Time` representing an absolute value on the timeline. A function `Clock` returns a value of type `Time` representing "now," along with overloaded addition and subtraction operators taking `Time` and `Time_Span` parameters. The package also provides operators for comparing `Time` values. (The value returned by `Clock` is monotonically increasing so you don't need to handle time zone jumps and other such things, unlike the function provided by `Ada.Calendar`.)

If the timeout is not context-specific then we'd use a constant as we did above, otherwise we'd allow the caller to specify the timeout. For example, here's a polling routine included with the DMA device driver we've mentioned a few times now. Some device-specific parts have been removed to keep the example simple. The appropriate timeout varies, so it is a parameter to the call:

```
procedure Poll_For_Completion
  (This      : in out DMA_Controller;
   Stream    : DMA_Stream_Selector;
   Timeout   : Time_Span;
   Result    : out DMA_Error_Code)
is
  Deadline : constant Time := Clock + Timeout;
begin
  Result := DMA_No_Error;  -- initially
  Polling : loop
    exit Polling when Status (This, Stream, Transfer_Complete_Indicated);
    if Clock >= Deadline then
      Result := DMA_Timeout_Error;
      return;
    end if;
  end loop Polling;
  Clear_Status (This, Stream, Transfer_Complete_Indicated);
end Poll_For_Completion;
```

In this approach, we compute the deadline as a point on the timeline by adding the value returned from the `Clock` function (i.e., "now") to the time interval specified by the parameter. Then, within the loop, we compare the value of the `Clock` to that deadline.

Finally, with another design approach we can reduce the processor cycles "wasted" when the polled device is not yet ready. Specifically, in the polling loop, when the device has not yet completed the requested function, we can temporarily relinquish the processor so that other tasks within the application can execute. That isn't perfect because we're still checking the device status even though we cannot exit the loop. And it requires other tasks to exist in your design, although that's probably a good idea for other reasons (e.g., logical threads having different, non-harmonic periods). This approach would look like this (an incomplete example):

```
procedure Poll_With_Delay is
  Next_Release : Time;
  Period       : constant Time_Span := Milliseconds (30); -- let's say
begin
  Next_Release := Clock;
  loop
    exit when Status (...);
```

(continues on next page)

(continued from previous page)

```

Next_Release := Next_Release + Period;
  delay until Next_Release;
end loop;
end Poll_With_Delay;
```

The code above will check the status of some device every 30 milliseconds (an arbitrary period just for illustration) until the Status function result allows the loop to exit. If the device "hangs" the loop is never exited, but as you saw there are ways to address that possibility. When the code does not exit the loop, the next point on the timeline is computed and the task executing the code then suspends, allowing the other tasks in the application to execute. Eventually, the next release point is reached and so the task becomes ready to execute again (and will, subject to priorities).

But how long should the polling task suspend when awaiting the device? We need to suspend long enough for the other tasks to get something done, but not so long that the device isn't handled fast enough. Finding the right balance is often not simple, and is further complicated by the "task switching" time. That's the time it takes to switch the execution context from one task to another, in this case in response to the "delay until" statement suspending the polling task. And it must be considered in both directions: when the delay expires we'll eventually switch back to the polling task.

As you can see, polling is easily expressed but has potentially significant drawbacks and architectural ramifications so it should be avoided as a default approach.

Now let's explore the Ada interrupt facilities.

35.2 Language-Defined Interrupt Model

The Ada language standard defines a model for hardware interrupts, as well as language-defined mechanisms for handling interrupts consistent with that model. The model is defined in Annex C, the "Systems Programming" annex, section 3 "Interrupt Support." The following is the text of that section with only a few simplifications and elisions.

- Interrupts are said to occur. An occurrence of an interrupt is separable into generation and delivery.
 - Generation of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program.
 - Delivery is the action that invokes part of the program as response to the interrupt occurrence.
- Between generation and delivery, the interrupt occurrence is pending.
- Some or all interrupts may be blocked. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.
- Certain interrupts are reserved. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other RTL-defined means. The set of reserved interrupts is determined by the hardware and run-time library (RTL).
- Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be attached to that interrupt. The execution of that program unit, the interrupt handler, is invoked upon delivery of the interrupt occurrence.
- While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt.
- The corresponding interrupt is blocked while the handler executes. While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.

Whether such occurrences remain pending or are lost is determined by the hardware and the RTL.

- Each interrupt has a default treatment which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is defined by the RTL.
- An exception propagated from a handler that is invoked by an interrupt has no effect. In particular, it is not propagated out of the handler, in the same way that exceptions do not propagate outside of task bodies.
- If the Ceiling_Locking policy is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object. ("Protected object" is abbreviated as "PO" for convenience).
- If the hardware or the underlying system holds pending interrupt occurrences, the RTL must provide for later delivery of these occurrences to the program.

(The above is not everything in the model but we can ignore the rest in this introduction.)

Because interrupt occurrences are generated by the hardware and delivered by the underlying system software (run-time library or real-time operating system), the application code is mainly responsible for responding to occurrences. Of course, the application must first configure the relevant external devices so that they generate the expected interrupts.

The actual response is application-specific but is also hardware-specific. The latter often (but not always) requires clearing the interrupt status within the generating device so that the same occurrence is not delivered again.

Furthermore, the standard model requires the underlying software to block further occurrences while the handler executes, and only allow preemption by higher-priority interrupt occurrences (if any). The application handlers are not responsible for these semantics either. As you will see, the choice of program unit used for expressing handlers makes this all very convenient for the developer.

As a consequence, in terms of the response, the application developer must write the specific handlers and attach those handlers to the corresponding interrupts. Attaching the handlers is implemented in the underlying system software, and it is this same underlying software that delivers the occurrences.

We will now explore the Ada facilities in detail. At the end of this chapter we will explore some common idioms using these mechanisms, especially with regard to the handlers' interaction with the rest of the application.

35.3 Interrupt Handlers

Interrupt handling is, by definition, asynchronous: some event occurs that causes the processor to suspend the application, respond to the event, and then resume application execution.

Because these events are asynchronous, the actions performed by the interrupt handler and the application are subject to the same sorts of race conditions as multiple tasks acting on shared data.

For example, a "reader" task may be in the act of reading (copying) the value of some shared variable, only to be preempted by a "writer" task that updates the value of the variable. In that case, when the "reader" task resumes execution, it will finish the read operation but will, as a result, have a value that is partly from the old value and partly from the new value. The effect is unpredictable. An interrupt handler can have the same effect on shared data as the preempting "writer" task that interrupts the "reader" task. This problem is possible for shared data of any type that is not atomically read or written. You can think of large record objects if that helps, but it even applies to some scalars.

That scenario applies even if no explicit tasks are declared in the application. That's because an implicit "environment task" is executing the main subprogram. In that case, the main subprogram is the entire application, but more typically some non-null application code is actively executing in one or more tasks.

But it's not just a matter of tasks. We said that interrupts usually have priorities. Typically that means a higher-priority interrupt will preempt the execution of the handler for a lower-priority interrupt. It's the same issue.

Furthermore, the fact that an interrupt has occurred needs to be communicated to the application, for example to say that updated data are available, perhaps a sensor reading or characters from a serial port. As we said above, we usually don't want to poll for that fact, so the application must be able to suspend until the event has occurred. Often we'll have a dedicated task within the application that suspends, rather than the entire application, but that's an application detail.

Ada's protected objects address all these asynchronous issues. Shared data declared within a protected object can be accessed only via protected procedures or protected entries, both of which execute with mutually exclusive access. Hence no race conditions are possible.

Here is an extremely simple, but realistic, example of a PO. This is not an interrupt handler example — we'll get to that — but it does show a shared variable and a protected procedure that executes with mutually exclusive access no matter how many tasks concurrently call it. The PO provides unique serial numbers.

```
protected Serial_Number is
    procedure Get_Next (Number : out Positive);
private
    Value : Positive := 1;
end Serial_Number;

protected body Serial_Number is
    procedure Get_Next (Number : out Positive) is
begin
    Number := Value;
    Value := Value + 1;
end Get_Next;
end Serial_Number;
```

Imagine there are multiple assembly lines creating devices of various sorts. Each device gets a unique serial number. These assembly lines run concurrently, so the calls to Get_Next occur concurrently. Without mutually exclusive access to the Value variable, multiple devices could get the same serial number.

Protected entries can suspend a caller until some condition is true; in this case, the fact that an interrupt has occurred and been handled. (As we will see, a protected entry is not the only way to synchronize with an accessing task, but it is the most robust and general.)

Here's an example of a PO with a protected entry:

```
protected type Persistent_Signal is
    entry Wait;
    procedure Send;
private
    Signal_Arrived : Boolean := False;
end Persistent_Signal;

protected body Persistent_Signal is
```

(continues on next page)

(continued from previous page)

```

entry Wait when Signal_Arrived is
begin
    Signal_Arrived := False;
end Wait;

procedure Send is
begin
    Signal_Arrived := True;
end Send;

end Persistent_Signal;

```

This is a PO providing a "Persistent Signal" abstraction. It allows a task to wait for a "signal" from another task. The signal is not lost if the receiving task is not already waiting, hence the term "persistent." Specifically, if `Signal_Arrived` is `False`, a caller to `Wait` will be suspended until `Signal_Arrived` becomes `True`. A caller to `Send` sets `Signal_Arrived` to `True`. If a caller to `Wait` was already present, suspended, it will be allowed to continue execution. If no caller was waiting, eventually some caller will arrive, find `Signal_Arrived` `True`, and will be allowed to continue. In either case, the `Signal_Arrived` flag will be set back to `False` before the `Wait` caller is released. Protected objects can have a priority assigned, similar to tasks, so they are integrated into the global priority semantics including interrupt priorities.

Therefore, in Ada an interrupt handler is a protected procedure declared within some protected object (PO). A given PO may handle more than one interrupt, and if so, may use one or more protected procedures to do so.

Interrupts can be attached to a protected procedure handler using a mechanism we'll discuss shortly. When the corresponding interrupt occurs, the attached handler is invoked. Any exceptions propagated by the handler's execution are ignored and do not go past the procedure.

While the protected procedure handler executes, the corresponding interrupt is blocked. As a consequence, another occurrence of that same interrupt will not preempt the handler's execution. However, if the hardware does not allow interrupts to be blocked, no blocking occurs and a subsequent occurrence would preempt the current execution of the handler. In that case, your handlers must be written with that possibility in mind. Most targets do block interrupts so we will assume that behavior in the following descriptions.

The standard mutually exclusive access provided to the execution of protected procedures and entries is enforced whether the "call" originates in hardware, via an interrupt, or in the application software, via some task. While any protected action in the PO executes, the corresponding interrupt is blocked, such that another occurrence will not preempt the execution of that actions' procedure or entry body execution in the PO.

On some processors blocked interrupts are lost, they do not persist. However, if the hardware can deliver an interrupt that had been blocked, the Systems Programming Annex requires the handler to be invoked again later, subject to the PO semantics described above.

The default treatment for a given interrupt depends on the RTL implementation. The default may be to jump immediately to system-defined handler that merely loops forever, thereby "hanging" the system and preventing any further execution of the application. On a bare-board target that would be a very common approach. Alternatively the default could be to ignore the interrupt entirely.

As mentioned earlier, some interrupts may be reserved, meaning that the application cannot install a replacement handler. For instance, most bare-board systems include a clock that is driven by a dedicated interrupt. The application cannot (or at least should not) override the interrupt handler for that interrupt. The determination of which interrupts are reserved is RTL-defined. Attempting to attach a user-defined handler for a reserved interrupt raises `Program_Error`, and the existing treatment is unchanged.

35.4 Interrupt Management

Ada defines a standard package that provides a primary type for identifying individual interrupts, as well as subprograms that take a parameter of that type in order to manage the system's interrupts and handlers. The package is named `Ada.Interrupts`, appropriately.

The primary type in that package is named `Interrupt_Id` and is a compiler-defined discrete type, meaning that it is either an integer type (signed or not) or an enumeration type. That representation is guaranteed so you can be sure that `Interrupt_Id` can be used, for example, as the index for an array type.

Package `Ada.Interrupts` provides functions to query whether a given interrupt is reserved, or if an interrupt has a handler attached. Procedures are defined to allow the application to attach and detach handlers, among other things. These procedures allow the application to dynamically manage interrupts. For example, when a new external device is added, perhaps as a "hot spare" replacing a damaged device, or when a new external device is simply connected to the target, the application can arrange to handle the new interrupts without having to recompile the application or restart application execution.

However, typically you will not use these procedures or functions to manage interrupts. In part that's because the architecture is usually static, i.e., the handlers are set up once and then never changed. In that case you won't need to query whether a given exception is reserved at run-time, or to check whether a handler is attached. You'd know that already, as part of the system architecture choices. For the same reasons, another mechanism for attaching handlers is more commonly used, and will be explained in that section. The package's type `Interrupt_Id`, however, will be used extensively.

A child package `Ada.Interrupts.Names` defines a target-dependent set of constants providing meaningful names for the `Interrupt_Id` values the target supports. Both the number of constants and their names are defined by the compiler, reflecting the variations in hardware available. This package and the enclosed constants are used all the time. For the sake of illustration, here is part of the package declaration for a Cortex M4F microcontroller supported by GNAT:

```
package Ada.Interrupts.Names is
  Sys_Tick_Interrupt      : constant Interrupt_ID := 1;
  ...
  EXTI0_Interrupt         : constant Interrupt_ID := 8;
  ...
  DMA1_Stream0_Interrupt : constant Interrupt_ID := 13;
  ...
  HASH_RNG_Interrupt     : constant Interrupt_ID := 80;
  ...
end Ada.Interrupts.Names;
```

Notice `HASH_RNG_Interrupt`, the name for `Interrupt_Id` value 80 on this target. That is the interrupt that the on-chip random number generator hardware uses to signal that a new value is available. We will use this interrupt in an example at the end of this chapter.

The representation chosen by the compiler for `Interrupt_Id` is very likely an integer, as in the above package, so the child package provides readable names for the numeric values. If `Interrupt_Id` is represented as an enumeration type the enumerational values are probably sufficiently readable, but the child package must be provided by the vendor nonetheless.

35.5 Associating Handlers With Interrupts

As we mentioned above, the Ada standard provides two ways to attach handlers to interrupts. One is procedural, described earlier. The other mechanism is automatic, achieved during elaboration of the protected object enclosing the handler procedure. The behavior is not unlike the activation of tasks: declared tasks are activated automatically as a result of their elaboration, whereas dynamically allocated tasks are activated as a result of their allocations.

We will focus exclusively on the automatic, elaboration-driven attachment model because that is the more common usage, and as a result, that is what GNAT supports on bare-board targets. It is also the mechanism that the standard Ravenscar and Jorvik profiles require. Our examples are consistent with those targets.

In the elaboration-based attachment model, we specify the interrupt to be attached to a given protected procedure within a protected object. This interrupt specification occurs within the enclosing protected object declaration. (Details in a moment.) When the enclosing PO is elaborated, the run-time library installs that procedure as the handler for that interrupt. A given PO may contain one or more interrupt handler procedures, as well as any other protected subprograms and entries.

In particular, we can associate an interrupt with a protected procedure by applying the aspect `Attach_Handler` to that procedure as part of its declaration, with the `Interrupt_Id` value as the aspect parameter. The association can also be achieved via a pragma with the same name as the aspect. Strictly speaking, the pragma `Attach_Handler` is obsolescent, but that just means that there is a newer way to make the association (i.e., the aspect). The pragma is not illegal and will remain supported. Because the pragma existed in a version of Ada prior to aspects you will see a lot of existing code using the pragma. You should become familiar with it. There's no language-driven reason to change the source code to use the aspect. New code should arguably use the aspect, but there's no technical reason to prefer one over the other.

Here is an example of a protected object with one protected procedure interrupt handler. It uses the `Attach_Handler` aspect to tie a random number generator interrupt to the `RNG_Controller.Interrupt_Handler` procedure:

```
protected RNG_Controller is
  ...
  entry Get_Random (Value : out UInt32);
private
  Last_Sample    : UInt32 := 0;
  Buffer         : Ring_Buffer;
  Data_Available : Boolean := False;

  procedure Interrupt_Handler with
    Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;
end RNG_Controller;
```

That's all that the developer must do to install the handler. The compiler and run-time library do the rest, automatically.

The local variables are declared in the private part, as required by the language, because they are shared data meant to be protected from race conditions. Therefore, the only compile-time access possible is via visible subprograms and entries declare in the visible part. Those subprograms and entries execute with mutually exclusive access so no race conditions are possible, as guaranteed by the language.

Note that procedure `Interrupt_Handler` is declared in the private part of `RNG_Controller`, rather than the visible part. That location is purely a matter of choice (unlike the variables),

but there is a good reason to hide it: application software can call an interrupt handler procedure too. If you don't ever intend for that to happen, have the compiler enforce your intent. An alert code reader will then recognize that clients cannot call that procedure. If, on the other hand, the handler is declared in the visible part, the reader must examine more of the code to determine whether there are any callers in the application code. Granted, a software call to an interrupt handler is rare, but not illegal, so you should state your intent in the code in an enforceable manner.

Be aware that the Ada compiler is allowed to place restrictions on protected procedure handlers. The compiler can restrict the content of the procedure body, for example, or it might forbid calls to the handler from the application software. The rationale is to allow direct invocation by the hardware, to minimize interrupt latency to the extent possible.

For completeness, here's the same RNG_Controller protected object using the pragma instead of the aspect to attach the interrupt to the handler procedure:

```
protected RNG_Controller is
  ...
  entry Get_Random (Value : out UInt32);
private

  Last_Sample    : UInt32 := 0;
  Buffer         : Ring_Buffer;
  Data_Available : Boolean := False;

  procedure Interrupt_Handler;
  pragma Attach_Handler (Interrupt_Handler,
                         Ada.Interrupts.Names.HASH_RNG_Interrupt);

end RNG_Controller;
```

As you can see, there isn't much difference. The aspect is somewhat more succinct. (The choice of where to declare the procedure remains the same.)

In this attachment model, protected declarations containing interrupt handlers must be declared at the library level. That means they must be declared in library packages. (Protected objects cannot be library units themselves, just as tasks cannot. They must be declared within some other unit.) Here is the full declaration for the RNG_Controller PO declared within a package — in this case within a package body:

```
with Ada.Interrupts.Names;
with Bounded_Ring_Buffers;

package body STM32.RNG.Interrupts is

  package UInt32_Buffers is new Bounded_Ring_Buffers (Content => UInt32);
  use UInt32_Buffers;

  protected RNG_Controller is
    ...
    entry Get_Random (Value : out UInt32);
  private

    Last_Sample    : UInt32 := 0;
    Samples        : Ring_Buffer (Upper_Bound => 9); -- arbitrary
    Data_Available : Boolean := False;

    procedure Interrupt_Handler with
      Attach_Handler => Ada.Interrupts.Names.HASH_RNG_Interrupt;

  end RNG_Controller;
```

(continues on next page)

(continued from previous page)

```
...  
end STM32.RNG.Interrupts;
```

But note that we're talking about protected declarations, a technical term that encompasses not only protected types but also anonymously-typed protected objects. In the `RNG_Controller` example, the PO does not have an explicit type declared; it is anonymously-typed. (Task objects can also be anonymously-typed.) You don't have to use a two-step process of first declaring the type and then an object of the type. If you only need one, no explicit type is required.

Although interrupt handler protected types must be declared at library level, the Ada model allows you to have an object of the type declared elsewhere, not necessarily at library level. However, note that the Ravenscar and Jorvik profiles require protected interrupt handler objects — anonymously-typed or not — to be declared at the library level too, for the sake of analysis. The profiles also require the elaboration-based attachment mechanism we have shown. For the sake of the widest applicability, and because with GNAT the most likely use-case involves either Ravenscar or Jorvik, we are following those restrictions in our examples.

35.6 Interrupt Priorities

Many (but not all) processors assign priorities to interrupts, with blocking and preemption among priorities of different levels, much like preemptive priority-based task semantics. Consequently, the priority semantics for interrupt handlers are as if a hardware "task," executing at an interrupt level priority, calls the protected procedure handler.

Interrupt handlers in Ada are protected procedures, which do not have priorities individually, but the enclosing protected object can be assigned a priority that will apply to the handler(s) when executing.

Therefore, protected objects can have priorities assigned using values of subtype `System.Interrupt_Priority`, which are high enough to require the blocking of one or more interrupts. The specific values among the priority subtypes are not standardized but the intent is that interrupt priorities are higher (more urgent) than non-interrupt priorities, as if they are declared like so in package `System`:

```
subtype Any_Priority is Integer range compiler-defined;  
  
subtype Priority is Any_Priority  
  range Any_Priority'First .. compiler-defined;  
  
subtype Interrupt_Priority is Any_Priority  
  range Priority'Last + 1 .. Any_Priority'Last;
```

For example, here are the subtype declarations in the GNAT compiler for an Arm Cortex M4 target:

```
subtype Any_Priority      is Integer      range 0 .. 255;  
subtype Priority        is Any_Priority range Any_Priority'First .. 240;  
subtype Interrupt_Priority is Any_Priority range  
  Priority'Last + 1 .. Any_Priority'Last;
```

Although the ranges are compiler-defined, when the Systems Programming Annex is implemented the range of `System.Interrupt_Priority` must include at least one value. Vendors are not required to have a distinct priority value in `Interrupt_Priority` for each

hardware interrupt possible on a given target. On a bare-metal target, they probably will have a one-to-one correspondence, but might not in a target with an RTOS or host OS.

A PO containing an interrupt handler procedure must be given a priority within the `Interrupt_Priority` subtype's range. To do so, we apply the aspect `Interrupt_Priority` to the PO. Perhaps confusingly, the aspect and the value's required subtype have the same name.

```
with Ada.Interrupts.Names;  use Ada.Interrupts.Names;
with System;               use System;

package Gyro_Interrupts is

    protected Handler with
        Interrupt_Priority => Interrupt_Priority'Last
    is
        private
            procedure IRQ_Handler;
            pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
    end Handler;

end Gyro_Interrupts;
```

The code above uses the highest (most urgent) interrupt priority value but some other value could be used instead, as long as it is in the `Interrupt_Priority` subtype's range. `Constraint_Error` is raised otherwise.

There is also an alternative pragma, now obsolescent, with the same name as the aspect and subtype. Here is an example:

```
with Ada.Interrupts.Names;  use Ada.Interrupts.Names;

package Gyro_Interrupts is

    protected Handler is
        pragma Interrupt_Priority (245);
    private
        procedure IRQ_Handler;
        pragma Attach_Handler (IRQ_Handler, EXTI2_Interrupt);
    end Handler;

end Gyro_Interrupts;
```

In the above we set the interrupt priority to 245, presumably a value conformant with this specific target. You should be familiar with this pragma too, because there is some much existing code using it. New code should use the aspect, ideally.

If we don't specify the priority for some protected object containing an interrupt handler (using either the pragma or the aspect), the initial priority of protected objects of that type is compiler-defined, but within the range of the subtype `Interrupt_Priority`. Generally speaking, you should specify the priorities per those of the interrupts handled, assuming they have distinct values, so that you can reason concretely about the relative blocking behavior at run-time.

Note that the parameter specifying the priority is optional for the `Interrupt_Priority` pragma. When none is given, the effect is as if the value `Interrupt_Priority'Last` was specified.

```
with Ada.Interrupts.Names;  use Ada.Interrupts.Names;

package Gyro_Interrupts is
```

(continues on next page)

(continued from previous page)

```
protected Handler is
    pragma Interrupt_Priority;
private
    ...
end Handler;

end Gyro_Interrupts;
```

No pragma parameter is given in the above, therefore `Gyro_Interrupts.Handler` executes at `Interrupt_Priority'Last` when invoked.

While an interrupt handler is executing, the corresponding interrupt is blocked. Therefore, the same interrupt will not be delivered again while the handler is executing. Plus, the protected object semantics mean that no software caller is also concurrently executing within the protected object. So no data race conditions are possible. If the system does not support blocking, however, the interrupt is not blocked when the handler executes.

In addition, when interrupt priorities are involved, hardware blocking typically extends to interrupts of equal or lower priority.

You should understand that a higher-priority interrupt could preempt the execution of a lower-priority interrupt's handler. Handlers do not define "critical sections" in which the processor cannot be preempted at all (other than the case of the highest priority interrupt).

Preemption does not cause data races, usually, because the typical case is to have a given protected object handle only one interrupt. It follows that only that one interrupt handler has visibility to the protected data in any given protected object, therefore only that one handler can update it. Any preempting handler would be in a different protected object, hence the preempting handler could not possibly update the data in the preempted handler's PO. No data race condition is possible.

However, protected objects can contain handlers for more than one interrupt. In that case, depending on the priorities, the execution of a higher-priority handler could preempt the execution of a lower priority handler in that same PO. Because each handler in the PO can update the local protected data, these data are effectively shared among asynchronous writers. Data race conditions are, as a result, possible.

The solution to the case of multiple handlers in a single PO is to assign the PO a priority not less than the highest of the interrupt priorities for which it contains handlers. That's known as the "ceiling priority" and works the same as when applying the ceiling for the priorities of caller tasks in the software. Then, whenever any interrupt handled by that PO is delivered, the handler executes at the ceiling priority, not necessarily the priority of the specific interrupt handled. All interrupts at a priority equal or lower than the PO priority are blocked, so no preemption by another handler within that same PO is possible. As a result, a handler for a higher priority interrupt must be in a different PO. If that higher priority handler is invoked, it can indeed preempt the execution of the handler for the lower priority interrupt in another PO. But because these two handlers will not be in the same PO, they will not share the data, so again no race condition is possible.

Note also that software callers will execute at the PO priority as well, so their priority may be increased during that execution. As you can see, the Ceiling Priority Protocol integrates application-level priorities, for tasks and protected objects, with interrupt-level priorities for interrupt handlers.

The Ceiling Locking Protocol is requested by specifying the `Ceiling_Locking` policy (see ARM D.3) to the pragma `Locking_Policy`. Both Ravenscar and Jorvik do so, automatically.

35.7 Common Design Idioms

In this section we explore some of the common idioms used when writing interrupt handlers in Ada.

35.7.1 Parameterizing Handlers

Suppose we have more than one instance of a kind of device. For example, multiple DMA controllers are often available on a System-on-Chip such as an Arm microcontroller. We can simplify our code by defining a device driver **type**, with one object of the type per supported hardware device. This is the same abstract data type (ADT) approach we'd take for software objects in application code, and in general for device drivers when multiple hardware instances are available.

We can also apply the ADT approach to interrupt handlers when we have multiple devices of a given kind that can generate interrupts. In this case, the type will be fully implemented as a protected type containing at least one interrupt handling procedure, with or without additional protected procedures or entries.

As is the case with abstract data types in general, we can tailor each object with discriminants defined with the type, in order to "parameterize" the type and thus allow distinct objects to have different characteristics. For example, we might define a bounded buffer ADT with a discriminant specifying the upper bound, so that distinct objects of the single type could have different bounds. In the case of hardware device instances, one of these parameters will often specify the device being driven, but we can also specify other device-specific characteristics. In particular, for interrupt handler types both the interrupt to handle and the interrupt priority can be discriminants. That's possible because the aspects/pragmas do not require their values to be specified via literals, unlike what was done in the `RNG_Controller` example above.

For example, here is the declaration for an interrupt handler ADT named `DMA_Interrupt_Controller`. This type manages the interrupts for a given DMA device, known as a `DMA_Controller`. Type `DMA_Controller` is itself an abstract data type, declared elsewhere.

```
protected type DMA_Interrupt_Controller
  (Controller : not null access DMA_Controller;
   Stream    : DMA_Stream_Selector;
   IRQ      : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is

  procedure Start_Transfer
    (Source    : Address;
     Destination : Address;
     Data_Count : UInt16);

  procedure Abort_Transfer (Result : out DMA_Error_Code);

  procedure Clear_Transfer_State;

  function Buffer_Error return Boolean;

  entry Wait_For_Completion (Status : out DMA_Error_Code);

private
```

(continues on next page)

(continued from previous page)

```
procedure Interrupt_Handler with Attach_Handler => IRQ;

No_Transfer_In_Progress : Boolean := True;
Last_Status : DMA_Error_Code := DMA_No_Error;
Had_Buffer_Error : Boolean := False;

end DMA_Interrupt_Controller;
```

In the above, the Controller discriminant provides an access value designating the specific DMA_Controller device instance to be managed. Each DMA device supports multiple independent conversion "streams" so the Stream discriminant specifies that characteristic. The IRQ and IRQ_Priority discriminants specify the handler values for that specific device and stream. These discriminant values are then used in the Interrupt_Priority pragma and the Attach_Handler aspect in the private part. ("IRQ" is a command handler name across programming languages, and is an abbreviation for "interrupt request.")

Here then are the declarations for two instances of the interrupt handler type:

```
DMA2_Stream0 : DMA_Interrupt_Controller
(Controller => DMA_2'Access,
 Stream     => Stream_0,
 IRQ        => DMA2_Stream0_Interrupt,
 IRQ_Priority => Interrupt_Priority'Last);

DMA2_Stream5 : DMA_Interrupt_Controller
(Controller => DMA_2'Access,
 Stream     => Stream_5,
 IRQ        => DMA2_Stream5_Interrupt,
 IRQ_Priority => Interrupt_Priority'Last);
```

In the above, both objects DMA2_Stream0 and DMA2_Stream5 are associated with the same object named DMA2, an instance of the DMA_Controller type. The difference in the objects is the stream that generates the interrupts they handle. One object handles Stream_0 interrupts and the other handles those from Stream_5. Package Ada.Interrupts.Names for this target (for GNAT) declares distinct names for the streams and devices generating the interrupts, hence DMA2_Stream0_Interrupt and DMA2_Stream5_Interrupt.

On both objects the priority is the highest interrupt priority (and hence the highest overall), Interrupt_Priority'Last. That will work, but of course all interrupts will be blocked during the execution of the handler, as well as the execution of any other subprogram or entry in the same PO. That means that the clock interrupt is blocked for that interval, for example. We use that interrupt value in our demonstrations for expedience, but in a real application you'd almost certainly use a lower value specific to the interrupt handled.

We could reduce the number of discriminants, and also make the code more robust, by taking advantage of the requirement that type Interrupt_Id be a discrete type. As such, it can be used as the index type into arrays. Here is a driver example with only the Interrupt_Id discriminant required:

```
Device_Priority : constant array (Interrupt_Id) of Interrupt_Priority := ( ... );

protected type Device_Interface
  (IRQ : Interrupt_Id)
with
  Interrupt_Priority => Device_Priority (IRQ)
is
  procedure Handler with Attach_Handler => IRQ;
  ...
end Device_Interface;
```

Now we use the one IRQ discriminant both to assign the priorities for distinct objects and

to attach their handler procedures.

35.7.2 Multi-Level Handlers

Interrupt handlers are intended to be very brief, in part because they prevent lower priority interrupts and application tasks from executing.

However, complete interrupt processing may require more than just the short protected procedure handler's activity. Therefore, two levels of handling are common: the protected procedure interrupt handler and a task. The handler does the least possible and then signals the task to do the rest.

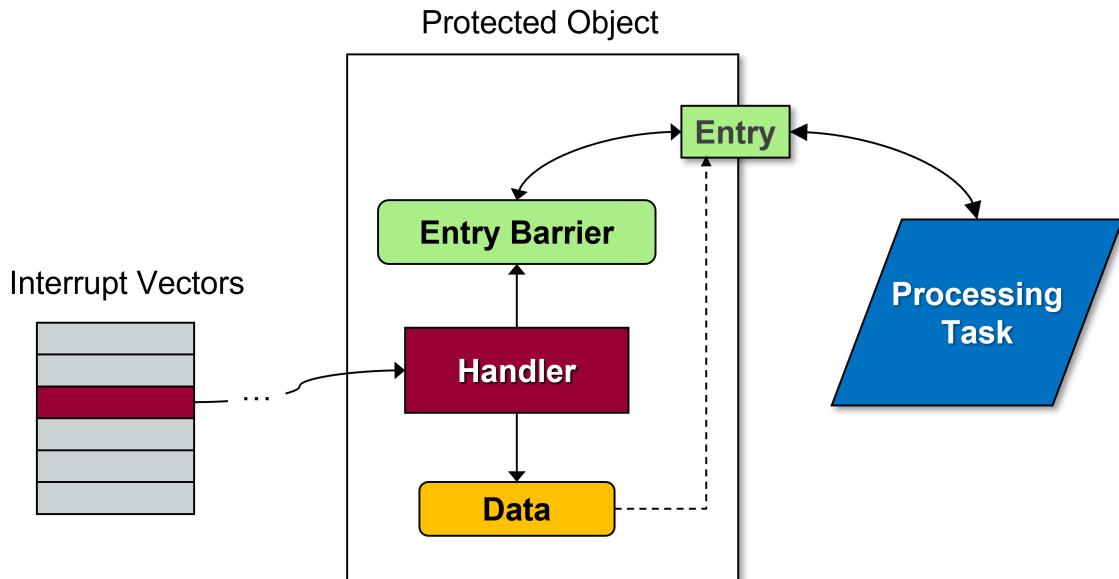
Of course, sometimes the handler does everything required and just needs to signal the application. In that case, the awakened task does no further "interrupt processing" but simply uses the result.

Regardless, the same issues apply: 1) How do application tasks synchronize with the handlers? Assuming the task is not polling the event, at some point the task must stop what it was doing and suspend, waiting for the handler to signal it. 2) Once synchronized, how can the handlers pass data to the tasks?

Using protected objects for interrupt handling provides an efficient mechanism that elegantly addresses both issues. In addition, when data communication is not required, another standard language mechanism is available. These give rise to two design idioms. We will explore both.

In the first idiom, the protected object contains a protected entry as well as the interrupt handler procedure. The task suspends on the entry when ready for the handler results, controlled by the barrier condition as usual. The protected handler procedure responds to interrupts, managing data (if any) as required. When ready, based on what the handler does, the handler sets the entry barrier to **True**. That allows the suspended task to execute the entry body. The entry body can do whatever is required, possibly just copying the local protected data to the entry parameters. Of course, the entry may be used purely for synchronizing with the handler, i.e., suspending and resuming the task, in which case there would be no parameters passed.

The image below depicts this design.



The DMA_Interrupt_Controller described earlier actually uses this design.

```

protected type DMA_Interrupt_Controller
  (Controller  : not null access DMA_Controller;
   Stream      : DMA_Stream_Selector;
   IRQ         : Ada.Interrupts.Interrupt_Id;
   IRQ_Priority : System Interrupt_Priority)
with
  Interrupt_Priority => IRQ_Priority
is

  procedure Start_Transfer
    (Source      : Address;
     Destination : Address;
     Data_Count  : UInt16);

  procedure Abort_Transfer (Result : out DMA_Error_Code);

  procedure Clear_Transfer_State;

  function Buffer_Error return Boolean;

  entry Wait_For_Completion (Status : out DMA_Error_Code);

private

  procedure Interrupt_Handler with Attach_Handler => IRQ;

  No_Transfer_In_Progress : Boolean := True;
  Last_Status             : DMA_Error_Code := DMA_No_Error;
  Had_Buffer_Error        : Boolean := False;

end DMA_Interrupt_Controller;

```

The client application code (task) calls procedure `Start_Transfer` to initiate the DMA transaction, then presumably goes off to accomplish something else, and eventually calls the `Wait_For_Completion` entry. That call blocks the task if the device has not yet completed the DMA transfer. The interrupt handler procedure, cleverly named `Interrupt_Handler`,

handles the interrupts, one of which indicates that the transfer has completed. Device errors also generate interrupts so the handler detects them and acts accordingly. Eventually, the handler sets the barrier to **True** and the task can get the status via the entry parameter.

```

procedure Start_Transfer
  (Source      : Address;
   Destination : Address;
   Data_Count  : UInt16)
is
begin
  No_Transfer_In_Progress := False;
  Had_Buffer_Error := False;
  Clear_All_Status (Controller.all, Stream);
  Start_Transfer_with_Interrupts
    (Controller.all,
     Stream,
     Source,
     . . .
     Enabled_Interrupts =>
       (Half_Transfer_Complete_Interrupt => False,
        others => True));
end Start_Transfer;

entry Wait_For_Completion
  (Status : out DMA_Error_Code)
when
  No_Transfer_In_Progress
is
begin
  Status := Last_Status;
end Wait_For_Completion;

```

In the above, the entry barrier consists of the Boolean variable `No_Transfer_In_Progress`. Procedure `Start_Transfer` first sets that variable to `False` so that a caller to `Wait_For_Completion` will suspend until the transaction completes one way or the other. Eventually, the handler sets `No_Transfer_In_Progress` to `True`.

```

procedure Interrupt_Handler is
  subtype Checked_Status_Flag is DMA_Status_Flag with
    Static_Predicate => Checked_Status_Flag /= Half_Transfer_Complete_Indicated;
begin
  for Flag in Checked_Status_Flag loop
    if Status (Controller.all, Stream, Flag) then
      case Flag is
        when FIFO_Error_Indicated =>
          Last_Status := DMA_FIFO_Error;
          Had_Buffer_Error := True;
          No_Transfer_In_Progress := not Enabled (Controller.all, Stream);
        when Direct_Mode_Error_Indicated =>
          Last_Status := DMA_Direct_Mode_Error;
          No_Transfer_In_Progress := not Enabled (Controller.all, Stream);
        when Transfer_Error_Indicated =>
          Last_Status := DMA_Transfer_Error;
          No_Transfer_In_Progress := True;
        when Transfer_Complete_Indicated =>
          Last_Status := DMA_No_Error;
          No_Transfer_In_Progress := True;
      end case;
      Clear_Status (Controller.all, Stream, Flag);
    end if;
  end loop;

```

(continues on next page)

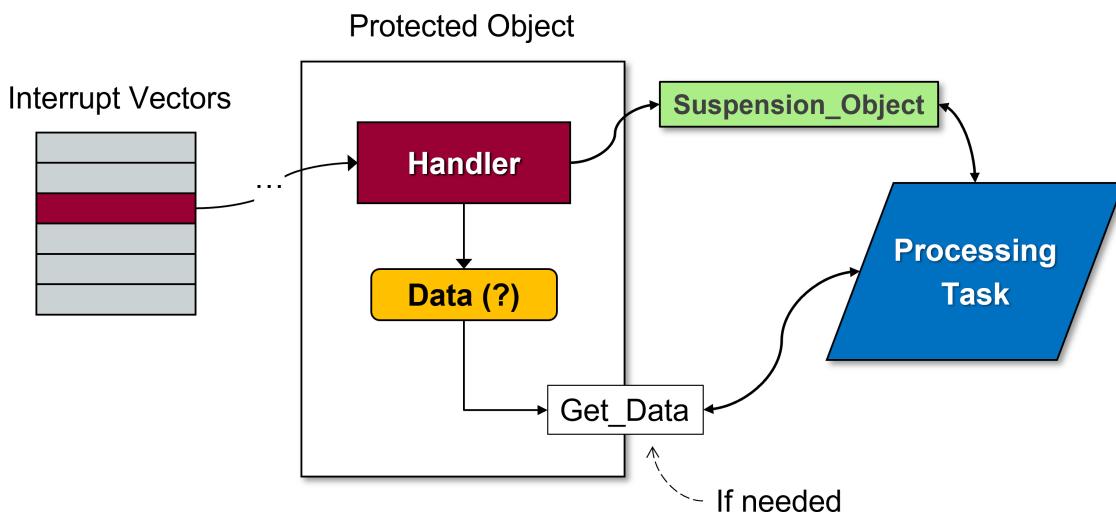
(continued from previous page)

```
end Interrupt_Handler;
```

This device driver doesn't bother with interrupts indicating that transfers are half-way complete so that specific status flag is ignored. In response to an interrupt, the handler checks each status flag to determine what happened. Note the resulting assignments for both the protected variables `Last_Status` and `No_Transfer_In_Progress`. The variable `No_Transfer_In_Progress` controls the entry, and `Last_Status` is passed to the caller via the entry formal parameter. When the interrupt handler exits, the resulting protected action allows the now-enabled entry call to execute.

In the second design idiom, the handler again synchronizes with the application task, but not using a protected entry.

The image below depicts this design.



In this approach, the task synchronizes with the handler using a `Suspension_Object` variable. The type `Suspension_Object` is defined in the language standard package Ada. `Synchronous_Task_Control`. Essentially, the type provides a thread-safe Boolean flag. Callers can suspend themselves (hence the package name) until another task resumes them by setting the flag to `True`. Here's the package declaration, somewhat elided:

```
package Ada.Synchronous_Task_Control is
    type Suspension_Object is limited private;
    procedure Set_True (S : in out Suspension_Object);
    procedure Set_False (S : in out Suspension_Object);
    function Current_State (S : Suspension_Object) return Boolean;
    procedure Suspend_Until_True (S : in out Suspension_Object);
private
    ...
end Ada.Synchronous_Task_Control;
```

Tasks call `Suspend_Until_True` to suspend themselves on some object of the type passed

as the parameter. The call suspends the caller until that object becomes **True**. If it is already **True**, the caller continues immediately. Objects of type `Suspension_Object` are automatically set to **False** initially, and become **True** via a call to `Set_True`. As part of the return from a call to `Suspend_Until_True`, the flag is set back to **False**. As a result, you probably only need those two subprograms.

The interrupt handler procedure responds to interrupts, eventually setting some visible `Suspension_Object` to **True** so that the caller will be signaled and resume. Here's an example showing both the protected object, with handler, and a `Suspension_Object` declaration:

```
with Ada.Interrupts.Names;           use Ada.Interrupts.Names;
with Ada.Synchronous_Task_Control;  use Ada.Synchronous_Task_Control;

package Gyro_Interrupts is

  Data_Available : Suspension_Object;

  protected Handler is
    pragma Interrupt_Priority;
  private
    procedure IRQ_Handler
      with Attach_Handler => EXTI2_Interrupt;
  end Handler;

end Gyro_Interrupts;
```

In the code above, `Gyro_Interrupts.Data_Available` is the `Suspension_Object` variable visible both to the interrupt handler PO and the client task.

`EXTI2_Interrupt` is "external interrupt number 2" on this particular microcontroller. It is connected to an external device, not on the SoC itself. Specifically, it is connected to a `L3GD20 MEMS motion sensor`⁵³, a three-axis digital output gyroscope. This gyroscope can be either polled or generate interrupts when ever data are available. The handler is very simple:

```
with STM32.EXTI;  use STM32.EXTI;

package body Gyro_Interrupts is

  protected body Handler is

    procedure IRQ_Handler is
    begin
      if External_Interrupt_Pending (EXTI_Line_2) then
        Clear_Exernal_Interrupt (EXTI_Line_2);
        Set_True (Data_Available);
      end if;
    end IRQ_Handler;

  end Handler;

end Gyro_Interrupts;
```

The handler simply clears the interrupt and resumes the caller task via a call to `Set_True` on the variable declared in the package spec.

The lack of an entry means that no data can be passed to the task via entry parameters. It is possible to pass data to the task but doing so would require an additional protected procedure or function.

⁵³ <https://www.st.com/en/mems-and-sensors/l3gd20.html>

The gyroscope hardware device interface is in package L3GD20. Here are the pertinent parts:

```
package L3GD20 is

    type Three_Axis_Gyroscope is tagged limited private;

    procedure Initialize
        (This      : in out Three_Axis_Gyroscope;
         Port      : Any_SPI_Port;
         Chip_Select : Any_GPIO_Point);

    ...

    procedure Enable_Data_Ready_Interrupt (This : in out Three_Axis_Gyroscope);

    ...

    type Angle_Rate is new Integer_16;

    type Angle_Rates is record
        X : Angle_Rate; -- pitch, per Figure 2, pg 7 of the Datasheet
        Y : Angle_Rate; -- roll
        Z : Angle_Rate; -- yaw
    end record with Size => 3 * 16;

    ...

    procedure Get_Raw_Angle_Rates
        (This  : Three_Axis_Gyroscope;
         Rates : out Angle_Rates);

    ...

end L3GD20;
```

With those packages available, we can write a simple main program to use the gyro. The real demo displayed the readings on an LCD but we've elided all those irrelevant details:

```
with Gyro_Interrupts;
with Ada.Synchronous_Task_Control;  use Ada.Synchronous_Task_Control;
with L3GD20;                      use L3GD20;
with STM32.Board;

procedure Demo_L3GD20 is
    Axes : L3GD20.Angle_Rates;

    ...

    procedure Await_Raw_Angle_Rates (Rates : out L3GD20.Angle_Rates) is
    begin
        Suspend_Until_True (Gyro_Interrupts.Data_Available);
        L3GD20.Get_Raw_Angle_Rates (STM32.Board.Gyro, Rates);
    end Await_Raw_Angle_Rates;

    ...

begin
    Configure_Gyro;
    Configure_Gyro_Interrupt;
```

(continues on next page)

(continued from previous page)

```

...
loop
    Await_Raw_Angle_Rates (Axes);
    ...
end loop;
end Demo_L3GD20;
```

The demo is a main procedure, even though we've been describing the client application code in terms of tasks. The main procedure is executed by the implicit "environment task" so it all still works. `Await_Raw_Angle_Rates` suspends (if necessary) on `Gyro_Interrupts`. `Data_Available` and then calls `L3GD20.Get_Raw_Angle_Rates` to get the rate values.

The operations provided by `Suspension_Object` are faster than protected entries, and noticeably so. However, that performance difference is due to the fact that `Suspension_Object` provides so much less capability than entries. In particular, there is no notion of protected actions, nor expressive entry barriers for condition synchronization, nor parameters to pass data while synchronized. Most importantly, there is no caller queue, so at most one caller can be waiting at a time on any given `Suspension_Object` variable. You'll get `Program_Error` if you try. Protected entries should be your first design choice. Note that the Ravenscar restrictions can make use of `Suspension_Object` much more likely.

35.8 Final Points

As you can see, the semantics of protected objects are a good fit for interrupt handling. However, other forms of handlers are allowed to be supported. For example, the compiler and RTL for a specific target may include support for interrupts generated by a device known to be available with that target. For illustration, let's imagine the target always has a serial port backed by a UART. In addition to handlers as protected procedure without parameters, perhaps the compiler and RTL support interrupt handlers with a single parameter of type `Unsigned_8` (or larger) as supported by the UART.

Overall, the interrupt model defined and supported by Ada is quite close to the canonical model presented by most programming languages, in part because it matches the model presented by typical hardware.

CHAPTER
THIRTYSIX

CONCLUSION

In the introduction to this course, we defined an "embedded system" as a computer that is part of a larger system, in which the capability to compute is not the larger system's primary function. These computers are said to be "embedded" in the larger system. That, in itself, sets this kind of programming apart from the more typical host-oriented programming. But the context also implies fewer resources are available, especially memory and electrical power, as well as processor power. Add to those limitations a frequent reliability requirement and you have a demanding context for development.

Using Ada can help you in this context, and for less cost than other languages, if you use it well. Many industrial organizations developing critical embedded software use Ada for that reason. Our goal in this course was to get you started in using it well.

To that end, we spent a lot of time talking about how to use Ada to do low level programming, such as how to specify the layout of types, how to map variables of those types to specific addresses, when and how to do unchecked programming (and how not to), and how to determine the validity of incoming data. Ada has a lot of support for this activity so there was much to explore.

Likewise, we examined development using Ada in combination with other languages, a not uncommon approach. Specifically, we saw how to interface with code and data written in other languages, and how (and why) to work with assembly language. Development in just one language is becoming less common over time so these were important aspects to know.

One of the more distinctive activities of embedded programming involves interacting with the outside world via embedded devices, such as A/D converters, timers, actuators, sensors, and so forth. (This can be one of the more entertaining activities as well.) We covered how to interact with these memory-mapped devices using representation specifications, data structures that simplified the functional code, and time-honored aspects of software engineering, including abstract data types.

Finally, we explored how to handle interrupts in Ada, another distinctive part of embedded systems programming. As we saw, Ada has extensive support for handling interrupts, using the same building blocks — protected objects — used in concurrent programming. These constructs provide a way to handle interrupts that is as portable as possible, in what is otherwise a very hardware-specific endeavor.

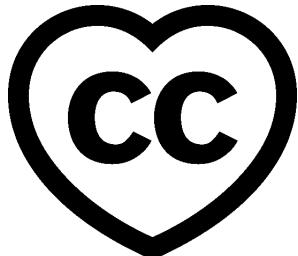
In the course, we mentioned a library of freely-available device drivers in Ada known as the Ada Driver Library (ADL). The ADL is a good resource for learning how Ada can be used to develop software for embedded systems using real-world devices and processors. Becoming familiar with it would be a good place to go next. Contributing to it would be even better! The ADL is available on GitHub for both non-proprietary and commercial use here: https://github.com/AdaCore/Ada_Drivers_Library.

Part IV

What's New in Ada 2022

Copyright © 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](#)⁵⁴



This course presents an overview of the new features of the latest Ada 2022 standard.

This document was written by Maxim Reznik and reviewed by Richard Kenner.

⁵⁴ <http://creativecommons.org/licenses/by-sa/4.0>

INTRODUCTION

This is a collection of short code examples demonstrating new features of the Ada 2022 Standard⁵⁵ as they are implemented in GNAT Ada compiler.

To use some of these features, you may need to use a compiler command line switch or pragma. Compilers starting with GNAT Community Edition 2021⁵⁶ or GCC 11⁵⁷ use **pragma Ada_2022**; or the -gnat2022 switch. Older compilers use **pragma Ada_2020**; or -gnat2020. To use the square brackets syntax or 'Reduce expressions, you need **pragma Extensions_Allowed** (On); or the -gnatX switch.

37.1 References

- Draft Ada 2022 Standard⁵⁸
- Ada 202x support in GNAT⁵⁹ blog post

⁵⁵ <http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html>

⁵⁶ <https://blog.adacore.com/gnat-community-2021-is-here>

⁵⁷ <https://gcc.gnu.org/gcc-11/>

⁵⁸ <http://www.ada-auth.org/standards/22aarm/html/AA-TTL.html>

⁵⁹ <https://blog.adacore.com/ada-202x-support-in-gnat>

'IMAGE ATTRIBUTE FOR ANY TYPE

Note: Attribute '`Image` for any type is supported by

- GNAT Community Edition 2020 and latter
 - GCC 11
-

38.1 'Image attribute for a value

Since the publication of the [Technical Corrigendum 1⁶⁰](#) in February 2016, the '`Image` attribute can now be applied to a value. So instead of `My_Type'Image (Value)`, you can just write `Value'Image`, as long as the Value is a name⁶¹. These two statements are equivalent:

```
Ada.Text_IO.Put_Line (Ada.Text_IO.Page_Length'Image);  
  
Ada.Text_IO.Put_Line  
  (Ada.Text_IO.Count'Image (Ada.Text_IO.Page_Length));
```

38.2 'Image attribute for any type

In Ada 2022, you can apply the '`Image` attribute to any type, including records, arrays, access types, and private types. Let's see how this works. We'll define array, record, and access types and corresponding objects and then convert these objects to strings and print them:

Listing 1: main.adb

```
1  pragma Ada_2022;  
2  
3  with Ada.Text_IO;  
4  
5  procedure Main is  
6    type Vector is array (Positive range <>) of Integer;  
7  
8    V1 : aliased Vector := [1, 2, 3];  
9  
10   type Text_Position is record  
11     Line, Column : Positive;  
12   end record;
```

(continues on next page)

⁶⁰ <https://reznikm.github.io/ada-auth/rm-4-NC/RM-0-1.html>

⁶¹ <https://reznikm.github.io/ada-auth/rm-4-NC/RM-4-1.html#S0091>

(continued from previous page)

```
13 Pos : constant Text_Position := (Line => 10, Column => 3);
14
15 type Vector_Access is access all Vector;
16
17 V1_Ptr : constant Vector_Access := V1'Access;
18
19 begin
20   Ada.Text_IO.Put_Line (V1'Image);
21   Ada.Text_IO.Put_Line (Pos'Image);
22   Ada.Text_IO.New_Line;
23   Ada.Text_IO.Put_Line (V1_Ptr'Image);
24 end Main;
```

Runtime output

```
[ 1, 2, 3]
(LINE => 10,
COLUMN => 3)
(access 7ffd3b4a7f8)
```

```
$ gprbuild -q -P main.gpr
Build completed successfully.
$ ./main
[ 1, 2, 3]
(LINE => 10,
COLUMN => 3)
(access 7fff64b23988)
```

Note the square brackets in the array image output. In Ada 2022, array aggregates could be written *this way* (page 525)!

38.3 References

- ARM 4.10 Image Attributes⁶²
- AI12-0020-1⁶³

⁶² <http://www.adu-auth.org/standards/22aarm/html/AA-4-10.html>

⁶³ <http://www.adu-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0020-1.txt>

REDEFINING THE 'IMAGE ATTRIBUTE

In Ada 2022, you can redefine '`Image`' attribute for your type, though the syntax to do this has been changed several times. Let's see how it works in GNAT Community 2021.

Note: Redefining attribute '`Image`' is supported by

- GNAT Community Edition 2021 (using `Text_Buffers`)
 - GNAT Community Edition 2020 (using `Text_Output_Utils`)
 - GCC 11 (using `Text_Output_Utils`)
-

In our example, let's redefine the '`Image`' attribute for a location in source code. To do this, we provide a new `Put_Image` aspect for the type:

Listing 1: main.adb

```
1  pragma Ada_2022;
2
3  with Ada.Text_IO;
4  with Ada.Strings.Text_Buffers;
5
6  procedure Main is
7
8    type Source_Location is record
9      Line   : Positive;
10     Column : Positive;
11   end record;
12   with Put_Image => My_Put_Image;
13
14  procedure My_Put_Image
15    (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
16     Value  : Source_Location);
17
18  procedure My_Put_Image
19    (Output : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
20     Value  : Source_Location)
21  is
22    Line   : constant String := Value.Line'Image;
23    Column : constant String := Value.Column'Image;
24    Result : constant String :=
25      Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
26  begin
27    Output.Put (Result);
28  end My_Put_Image;
29
30  Line_10 : constant Source_Location := (Line => 10, Column => 1);
31
32  begin
```

(continues on next page)

(continued from previous page)

```
33    Ada.Text_Io.Put_Line (Line_10'Image);
34 end Main;
```

Runtime output

```
10:1
```

39.1 What's the Root_Buffer_Type?

Let's see how it's defined in the Ada.Strings.Text_Buffers package.

```
type Root_Buffer_Type is abstract tagged limited private;

procedure Put
  (Buffer : in out Root_Buffer_Type;
   Item   : in     String) is abstract;
```

In addition to Put, there are also Wide_Put, Wide_Wide_Put, Put_UTF_8, Wide_Put_UTF_16. And also New_Line, Increase_Indent, Decrease_Indent.

39.2 Outdated draft implementation

GNAT Community Edition 2020 and GCC 11 both provide a draft implementation that's incompatible with the Ada 2022 specification. For those versions, My_Put_Image looks like:

```
procedure My_Put_Image
  (Sink  : in out Ada.Strings.Text_Output.Sink'Class;
   Value : Source_Location)
is
  Line   : constant String := Value.Line'Image;
  Column : constant String := Value.Column'Image;
  Result : constant String :=
    Line (2 .. Line'Last) & ':' & Column (2 .. Column'Last);
begin
  Ada.Strings.Text_Output_Utils.Put_UTF_8 (Sink, Result);
end My_Put_Image;
```

39.3 References

- ARM 4.10 Image Attributes⁶⁴
- AI12-0020-1⁶⁵
- AI12-0384-2⁶⁶

⁶⁴ <http://www.adা-auth.org/standards/22aarm/html/AA-4-10.html>

⁶⁵ <http://www.adা-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0020-1.TXT>

⁶⁶ <http://www.adা-auth.org/cgi-bin/cvsweb.cgi/ai12s/AI12-0384-2.TXT>

USER-DEFINED LITERALS

Note: User-defined literals are supported by

- GNAT Community Edition 2020
 - GCC 11
-

In Ada 2022, you can define string, integer, or real literals for your types. The compiler will convert such literals to your type at run time using a function you provide. To do so, specify one or more new aspects:

- Integer_Literal
- Real_Literal
- String_Literal

For our example, let's define all three for a simple type and see how they work. For simplicity, we use a `Wide_Wide_String` component for the internal representation:

Listing 1: main.adb

```
1  pragma Ada_2022;
2
3  with Ada.Wide_Wide_Text_IO;
4  with Ada.Characters.Conversions;
5
6  procedure Main is
7
8      type My_Type (Length : Natural) is record
9          Value : Wide_Wide_String (1 .. Length);
10     end record;
11    with String_Literal => From_String,
12        Real_Literal      => From_Real,
13        Integer_Literal   => From_Integer;
14
15    function From_String (Value : Wide_Wide_String) return My_Type is
16        ((Length => Value'Length, Value => Value));
17
18    function From_Real (Value : String) return My_Type is
19        ((Length => Value'Length,
20         Value  => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
21
22    function From_Integer (Value : String) return My_Type renames From_Real;
23
24    procedure Print (Self : My_Type) is
25    begin
26        Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
27    end Print;
```

(continues on next page)

(continued from previous page)

```

28
29 begin
30   Print ("Test ""string""");
31   Print (123);
32   Print (16#DEAD_BEEF#);
33   Print (2.99_792_458e+8);
34 end Main;

```

Runtime output

```

Test "string"
123
16#DEAD_BEEF#
2.99_792_458e+8

```

As you see, real and integer literals are converted to strings while preserving the formatting in the source code, while string literals are decoded: `From_String` is passed the specified string value. In all cases, the compiler translates these literals into function calls.

40.1 Turn Ada into JavaScript

Do you know that `'5'+3` in JavaScript is `53`?

```

> '5'+3
'53'

```

Now we can get the same result in Ada! But before we do, we need to define a custom `+` operator:

Listing 2: main.adb

```

1  pragma Ada_2022;
2
3  with Ada.Wide_Wide_Text_IO;
4  with Ada.Characters.Conversions;
5
6  procedure Main is
7
8    type My_Type (Length : Natural) is record
9      Value : Wide_Wide_String (1 .. Length);
10   end record;
11  with String_Literal => From_String,
12    Real_Literal     => From_Real,
13    Integer_Literal  => From_Integer;
14
15  function "+" (Left, Right : My_Type) return My_Type is
16    (Left.Length + Right.Length, Left.Value & Right.Value);
17
18  function From_String (Value : Wide_Wide_String) return My_Type is
19    ((Length => Value'Length, Value => Value));
20
21  function From_Real (Value : String) return My_Type is
22    ((Length => Value'Length,
23     Value  => Ada.Characters.Conversions.To_Wide_Wide_String (Value)));
24
25  function From_Integer (Value : String) return My_Type renames From_Real;
26
27  procedure Print (Self : My_Type) is

```

(continues on next page)

(continued from previous page)

```
28 begin
29   Ada.Wide_Wide_Text_IO.Put_Line (Self.Value);
30 end Print;
31
32 begin
33   Print ("5" & 3);
34 end Main;
```

Runtime output

```
53
```

Jokes aside, this feature is very useful. For example it allows a "native-looking API" for *big integers* (page 545).

40.2 References

- ARM 4.2.1 User-Defined Literals⁶⁷
- AI12-0249-1⁶⁸
- AI12-0342-1⁶⁹

⁶⁷ <http://www.adা-auth.org/standards/22rm/html/RM-4-2-1.html>

⁶⁸ <http://www.adা-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0249-1.TXT>

⁶⁹ <http://www.adা-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0342-1.TXT>

ADVANCED ARRAY AGGREGATES

Note: These array aggregates are supported by

- GNAT Community Edition 2020
 - GCC 11
-

41.1 Square brackets

In Ada 2022, you can use square brackets in array aggregates. Using square brackets simplifies writing both empty aggregates and single-element aggregates. Consider this:

Listing 1: show_square_brackets.ads

```
1  pragma Ada_2022;
2  pragma Extensions_Allowed (On);
3
4  package Show_Square_Brackets is
5
6    type Integer_Array is array (Positive range <>) of Integer;
7
8    Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
9    New_Style_Empty : Integer_Array := [];
10
11   Old_Style_One_Item : Integer_Array := (1 => 5);
12   New_Style_One_Item : Integer_Array := [5];
13
14 end Show_Square_Brackets;
```

Build output

```
show_square_brackets.ads:8:39: warning: array aggregate using () is an obsolescent
syntax, use [] instead [-gnatwj]
show_square_brackets.ads:11:42: warning: array aggregate using () is an
obsolescent syntax, use [] instead [-gnatwj]
```

Short summary for parentheses and brackets

- Record aggregates use parentheses
 - *Container aggregates* (page 529) use square brackets
 - Array aggregates can use both square brackets and parentheses, but parentheses usage is obsolescent
-

41.2 Iterated Component Association

There is a new kind of component association:

```
Vector : Integer_Array := [for J in 1 .. 5 => J * 2];
```

This association starts with **for** keyword, just like a quantified expression. It declares an index parameter that you can use in the computation of a component.

Iterated component associations can nest and can be nested in another association (iterated or not). Here we use this to define a square matrix:

```
Matrix : array (1 .. 3, 1 .. 3) of Positive :=
  [for J in 1 .. 3 =>
    [for K in 1 .. 3 => J * 10 + K]];
```

Iterated component associations in this form provide both element indices and values, just like named component associations:

```
Data : Integer_Array (1 .. 5) :=
  [for J in 2 .. 3 => J, 5 => 5, others => 0];
```

Here Data contains (0, 2, 3, 0, 5), not (2, 3, 5, 0, 0).

Another form of iterated component association corresponds to a positional component association and provides just values, but no element indices:

```
Vector_2 : Integer_Array := [for X of Vector => X / 2];
```

You cannot mix these forms in a single aggregate.

It's interesting that such aggregates were originally proposed more than 25 years ago!

Complete code snippet:

Listing 2: show_iterated_component_association.adb

```

1  pragma Ada_2022;
2  pragma Extensions_Allowed (On); -- for square brackets
3
4  with Ada.Text_IO;
5
6  procedure Show_Iterated_Component_Association is
7
8    type Integer_Array is array (Positive range <>) of Integer;
9
10   Old_Style_Empty : Integer_Array := (1 .. 0 => <>);
11   New_Style_Empty : Integer_Array := [];
12
13   Old_Style_One_Item : Integer_Array := (1 => 5);
14   New_Style_One_Item : Integer_Array := [5];
15
16   Vector : constant Integer_Array := [for J in 1 .. 5 => J * 2];
17
18   Matrix : constant array (1 .. 3, 1 .. 3) of Positive :=
19     [for J in 1 .. 3 =>
20      [for K in 1 .. 3 => J * 10 + K]];
21
22   Data : constant Integer_Array (1 .. 5) :=
23     [for J in 2 .. 3 => J, 5 => 5, others => 0];
24
25   Vector_2 : constant Integer_Array := [for X of Vector => X / 2];

```

(continues on next page)

(continued from previous page)

```

26 begin
27   Ada.Text_Io.Put_Line (Vector'Image);
28   Ada.Text_Io.Put_Line (Matrix'Image);
29   Ada.Text_Io.Put_Line (Data'Image);
30   Ada.Text_Io.Put_Line (Vector_2'Image);
31 end Show_Iterated_Component_Association;

```

Build output

```

show_iterated_component_association.adb:10:04: warning: variable "Old_Style_Empty"
  ↪is not referenced [-gnatwu]
show_iterated_component_association.adb:10:39: warning: array aggregate using () ↪
  ↪is an obsolescent syntax, use [] instead [-gnatwj]
show_iterated_component_association.adb:11:04: warning: variable "New_Style_Empty" ↪
  ↪is not referenced [-gnatwu]
show_iterated_component_association.adb:13:04: warning: variable "Old_Style_One_
  ↪Item" is not referenced [-gnatwu]
show_iterated_component_association.adb:13:42: warning: array aggregate using () ↪
  ↪is an obsolescent syntax, use [] instead [-gnatwj]
show_iterated_component_association.adb:14:04: warning: variable "New_Style_One_
  ↪Item" is not referenced [-gnatwu]

```

Runtime output

```

[ 2,  4,  6,  8, 10]

[
  [ 11, 12, 13],
  [ 21, 22, 23],
  [ 31, 32, 33]]
[ 0,  2,  3,  0,  5]
[ 1,  2,  3,  4,  5]

```

41.3 References

- ARM 4.3.3 Array Aggregates⁷⁰
- AI12-0212-1⁷¹
- AI12-0306-1⁷²

⁷⁰ <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-3.html>

⁷¹ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT>

⁷² <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0306-1.TXT>

CONTAINER AGGREGATES

Note: Container aggregates are supported by

- GNAT Community Edition 2021
 - GCC 11
-

Ada 2022 introduces container aggregates, which can be used to easily create values for vectors, lists, maps, and other aggregates. For containers such as maps, the aggregate must use named associations to provide keys and values. For other containers it uses positional associations. Only square brackets are allowed. Here's an example:

Listing 1: main.adb

```
1  pragma Ada_2022;
2
3  with Ada.Text_Io;
4  with Ada.Containers.Vectors;
5  with Ada.Containers.Ordered_Maps;
6
7  procedure Main is
8
9    package Int_Vectors is new Ada.Containers.Vectors
10   (Positive, Integer);
11
12   X : constant Int_Vectors.Vector := [1, 2, 3];
13
14   package Float_Maps is new Ada.Containers.Ordered_Maps
15   (Integer, Float);
16
17   Y : constant Float_Maps.Map := [-10 => 1.0, 0 => 2.5, 10 => 5.51];
18 begin
19   Ada.Text_Io.Put_Line (X'Image);
20   Ada.Text_Io.Put_Line (Y'Image);
21 end Main;
```

Runtime output

```
[ 1, 2, 3]
[-10 => 1.00000E+00, 0 => 2.50000E+00, 10 => 5.51000E+00]
```

At run time, the compiler creates an empty container and populates it with elements one by one. If you define a new container type, you can specify a new Aggregate aspect to enable container aggregates for your container and let the compiler know what subprograms to use to construct the aggregate:

Listing 2: main.adb

```

1  pragma Ada_2022;
2
3  procedure Main is
4
5    package JSON is
6      type JSON_Value is private
7        with Integer_Literal => To_JSON_Value;
8
9      function To_JSON_Value (Text : String) return JSON_Value;
10
11     type JSON_Array is private
12       with Aggregate => (Empty          => New_JSON_Array,
13                           Add_Unnamed => Append);
14
15     function New_JSON_Array return JSON_Array;
16
17   procedure Append
18     (Self  : in out JSON_Array;
19      Value : JSON_Value) is null;
20
21  private
22    type JSON_Value is null record;
23    type JSON_Array is null record;
24
25    function To_JSON_Value (Text : String) return JSON_Value
26      is (null record);
27
28    function New_JSON_Array return JSON_Array is (null record);
29  end JSON;
30
31  List : JSON.JSON_Array := [1, 2, 3];
32  -----
33 begin
34  -- Equivalent old initialization code
35  List := JSON.New_JSON_Array;
36  JSON.Append (List, 1);
37  JSON.Append (List, 2);
38  JSON.Append (List, 3);
39 end Main;

```

The equivalent for maps is:

Listing 3: main.adb

```

1  pragma Ada_2022;
2
3  procedure Main is
4
5    package JSON is
6      type JSON_Value is private
7        with Integer_Literal => To_JSON_Value;
8
9      function To_JSON_Value (Text : String) return JSON_Value;
10
11     type JSON_Object is private
12       with Aggregate => (Empty          => New_JSON_Object,
13                           Add_Named  => Insert);
14
15     function New_JSON_Object return JSON_Object;
16

```

(continues on next page)

(continued from previous page)

```

17  procedure Insert
18    (Self : in out JSON_Object;
19     Key  : Wide_Wide_String;
20     Value : JSON_Value) is null;
21
22  private
23    type JSON_Value is null record;
24    type JSON_Object is null record;
25
26    function To_JSON_Value (Text : String) return JSON_Value
27      is (null record);
28
29    function New_JSON_Object return JSON_Object is (null record);
30  end JSON;
31
32  Object : JSON.JSON_Object := ["a" => 1, "b" => 2, "c" => 3];
33
34 begin
35  -- Equivalent old initialization code
36  Object := JSON.New_JSON_Object;
37  JSON.Insert (Object, "a", 1);
38  JSON.Insert (Object, "b", 2);
39  JSON.Insert (Object, "c", 3);
40 end Main;

```

You can't specify both Add_Named and Add_Unnamed subprograms for the same type. This prevents you from defining JSON_Value with both array and object aggregates present. But we can define conversion functions for array and object and get code almost as dense as the same code in native JSON. For example:

Listing 4: main.adb

```

1  pragma Ada_2022;
2
3  procedure Main is
4
5    package JSON is
6      type JSON_Value is private
7        with Integer_Literal => To_Value, String_Literal => To_Value;
8
9      function To_Value (Text : String) return JSON_Value;
10     function To_Value (Text : Wide_Wide_String) return JSON_Value;
11
12    type JSON_Object is private
13      with Aggregate => (Empty      => New_JSON_Object,
14                           Add_Named => Insert);
15
16    function New_JSON_Object return JSON_Object;
17
18    procedure Insert
19      (Self : in out JSON_Object;
20       Key  : Wide_Wide_String;
21       Value : JSON_Value) is null;
22
23    function From_Object (Self : JSON_Object) return JSON_Value;
24
25    type JSON_Array is private
26      with Aggregate => (Empty      => New_JSON_Array,
27                           Add_Unnamed => Append);
28
29    function New_JSON_Array return JSON_Array;

```

(continues on next page)

(continued from previous page)

```

30
31      procedure Append
32          (Self : in out JSON_Array;
33           Value : JSON_Value) is null;
34
35      function From_Array (Self : JSON_Array) return JSON_Value;
36
37  private
38      type JSON_Value is null record;
39      type JSON_Object is null record;
40      type JSON_Array is null record;
41
42      function To_Value (Text : String) return JSON_Value is
43          (null record);
44      function To_Value (Text : Wide_Wide_String) return JSON_Value is
45          (null record);
46      function New_JSON_Object return JSON_Object is
47          (null record);
48      function New_JSON_Array return JSON_Array is
49          (null record);
50      function From_Object (Self : JSON_Object) return JSON_Value is
51          (null record);
52      function From_Array (Self : JSON_Array) return JSON_Value is
53          (null record);
54  end JSON;
55
56  function "+" (X : JSON.JSON_Object) return JSON.JSON_Value
57  renames JSON.From_Object;
58  function "-" (X : JSON.JSON_Array) return JSON.JSON_Value
59  renames JSON.From_Array;
60
61  Offices : JSON.JSON_Array :=
62      [+["name" => "North American Office",
63          "phones" => -[1_877_787_4628,
64                      1_866_787_4232,
65                      1_212_620_7300],
66          "email" => "info@adacore.com"],
67      +["name" => "European Office",
68          "phones" => -[33_1_49_70_67_16,
69                      33_1_49_70_05_52],
70          "email" => "info@adacore.com"]];
71  -----
72 begin
73  -- Equivalent old initialization code is too long to print it here
74  null;
75 end Main;

```

Build output

```
main.adb:61:04: warning: variable "Offices" is not referenced [-gnatwu]
```

The Offices variable is supposed to contain this value:

```
[{"name" : "North American Office",
 "phones": [18777874628,
            18667874232,
            12126207300],
 "email" : "info@adacore.com"}, {"name" : "European Office",
 "phones": [33149706716,
            33149700552],
```

(continues on next page)

(continued from previous page)

```
"email" : "info@adacore.com"}]
```

42.1 References

- ARM 4.3.5 Container Aggregates⁷³
- AI12-0212-1⁷⁴

⁷³ <http://www.adা-auth.org/standards/22aarm/html/AA-4-3-5.html>

⁷⁴ <http://www.adা-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0212-1.TXT>

DELTA AGGREGATES

Note: Delta aggregates are supported by

- GNAT Community Edition 2019
 - GCC 9
-

Sometimes you need to create a copy of an object, but with a few modifications. Before Ada 2022, doing this involves a dummy object declaration or an aggregate with associations for each property. The dummy object approach doesn't work in contract aspects or when there are limited components. On the other hand, re-listing properties in a large aggregate can be very tedious and error-prone. So, in Ada 2022, you can use a *delta aggregate* instead.

43.1 Delta aggregate for records

The delta aggregate for a record type looks like this:

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

Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);

Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
```

The more components you have, the more you will like the delta aggregate.

43.2 Delta aggregate for arrays

You can also use delta aggregates for arrays to change elements, but not bounds. Moreover, it only works for one-dimensional arrays of non-limited components.

```
type Vector_3D is array (1 .. 3) of Float;

Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
```

You can use parentheses for array aggregates, but you can't use square brackets for record aggregates.

Here is the complete code snippet:

Listing 1: main.adb

```
1  pragma Ada_2022;
2
3  with Ada.Text_IO;
4
5  procedure Main is
6
7    type Vector is record
8      X, Y, Z : Float;
9    end record;
10
11   Point_1 : constant Vector := (X => 1.0, Y => 2.0, Z => 3.0);
12   Projection_1 : constant Vector := (Point_1 with delta Z => 0.0);
13
14   type Vector_3D is array (1 .. 3) of Float;
15
16   Point_2 : constant Vector_3D := [1.0, 2.0, 3.0];
17   Projection_2 : constant Vector_3D := [Point_2 with delta 3 => 0.0];
18 begin
19   Ada.Text_IO.Put (Float'Image (Projection_1.X));
20   Ada.Text_IO.Put (Float'Image (Projection_1.Y));
21   Ada.Text_IO.Put (Float'Image (Projection_1.Z));
22   Ada.Text_IO.New_Line;
23   Ada.Text_IO.Put (Float'Image (Projection_2 (1)));
24   Ada.Text_IO.Put (Float'Image (Projection_2 (2)));
25   Ada.Text_IO.Put (Float'Image (Projection_2 (3)));
26   Ada.Text_IO.New_Line;
27 end Main;
```

Runtime output

```
1.00000E+00 2.00000E+00 0.00000E+00
1.00000E+00 2.00000E+00 0.00000E+00
```

43.3 References

- ARM 4.3.4 Delta Aggregates⁷⁵
- AI12-0127-1⁷⁶

⁷⁵ <http://www.ada-auth.org/standards/22aarm/html/AA-4-3-4.html>

⁷⁶ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0127-1.TXT>

CHAPTER
FORTYFOUR

TARGET NAME SYMBOL (@)

Note: Target name symbol is supported by

- GNAT Community Edition 2019
 - GCC 9
-

Ada 2022 introduces a new symbol, @, which can only appear on the right hand side of an assignment statement. This symbol acts as the equivalent of the name on the left hand side of that assignment statement. It was introduced to avoid code duplication: instead of retyping a (potentially long) name, you can use @. This symbol denotes a constant, so you can't pass it into [**in**] **out** arguments of a subprogram.

As an example, let's calculate some statistics for My_Data array:

Listing 1: statistics.ads

```
1  pragma Ada_2022;
2
3  package Statistics is
4
5      type Statistic is record
6          Count : Natural := 0;
7          Total : Float := 0.0;
8      end record;
9
10     My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
11
12     Statistic_For_My_Data : Statistic;
13
14 end Statistics;
```

To do this, we loop over My_Data elements:

Listing 2: main.adb

```
1  pragma Ada_2022;
2  with Ada.Text_IO;
3
4  procedure Main is
5
6      type Statistic is record
7          Count : Natural := 0;
8          Total : Float := 0.0;
9      end record;
10
11     My_Data : constant array (1 .. 5) of Float :=
12         [for J in 1 .. 5 => Float (J)];
```

(continues on next page)

(continued from previous page)

```
13      Statistic_For_My_Data : Statistic;
14
15  begin
16    for Data of My_Data loop
17      Statistic_For_My_Data.Count := @ + 1;
18      Statistic_For_My_Data.Total := @ + Data;
19    end loop;
20
21
22    Ada.Text_Io.Put_Line (Statistic_For_My_Data'Image);
23  end Main;
```

Runtime output

```
(COUNT => 5,
 TOTAL => 1.50000E+01)
```

Each right hand side is evaluated only once, no matter how many @ symbols it contains. Let's verify this by introducing a function call that prints a line each time it's called:

Listing 3: main.adb

```
1  pragma Ada_2022;
2  with Ada.Text_Io;
3
4  procedure Main is
5
6    My_Data : array (1 .. 5) of Float := [for J in 1 .. 5 => Float (J)];
7
8    function To_Index (Value : Positive) return Positive is
9    begin
10      Ada.Text_Io.Put_Line ("To_Index is called.");
11      return Value;
12    end To_Index;
13
14  begin
15    My_Data (To_Index (1)) := @ ** 2 - 3.0 * @;
16    Ada.Text_Io.Put_Line (My_Data'Image);
17  end Main;
```

Runtime output

```
To_Index is called.

[-2.00000E+00,  2.00000E+00,  3.00000E+00,  4.00000E+00,  5.00000E+00]
```

This use of @ may look a bit cryptic, but it's the best solution that was found. Unlike other languages (e.g., sum += x; in C), this approach lets you use @ an arbitrary number of times within the right hand side of an assignment statement.

44.1 Alternatives

In C++, the previous statement could be written with a reference type (one line longer!):

```
auto& a = my_data[to_index(1)];
a = a * a - 3.0 * a;
```

In Ada 2022, you can use a similar renaming:

```
declare
  A renames My_Data (To_Index (1));
begin
  A := A ** 2 - 3.0 * A;
end;
```

Here we use a new short form of the rename declaration, but this still looks too heavy, and even worse, it can't be used for discriminant-dependent components.

44.2 References

- ARM 5.2.1 Target Name Symbols⁷⁷
- AI12-0125-3⁷⁸

⁷⁷ <http://www.ada-auth.org/standards/22aarm/html/AA-5-2-1.html>

⁷⁸ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0125-3.TXT>

ENUMERATION REPRESENTATION

Note: Enumeration representation attributes are supported by

- GNAT Community Edition 2019
 - GCC 9
-

Enumeration types in Ada are represented as integers at the machine level. But there are actually two mappings from enumeration to integer: a literal position and a representation value.

45.1 Literal positions

Each enumeration literal has a corresponding position in the type declaration. We can easily obtain it from the **Type'Pos** (Enum) attribute.

Listing 1: main.adb

```
1 with Ada.Text_Io;
2 with Ada.Integer_Text_Io;
3
4 procedure Main is
5 begin
6   Ada.Text_Io.Put ("Pos(False) =");
7   Ada.Integer_Text_Io.Put (Boolean'Pos (False));
8   Ada.Text_Io.New_Line;
9   Ada.Text_Io.Put ("Pos(True) =");
10  Ada.Integer_Text_Io.Put (Boolean'Pos (True));
11 end Main;
```

Runtime output

```
Pos(False) =      0
Pos(True)  =      1
```

For the reverse mapping, we use **Type'Val** (Int):

Listing 2: main.adb

```
1 with Ada.Text_Io;
2
3 procedure Main is
4 begin
5   Ada.Text_Io.Put_Line (Boolean'Val (0)'Image);
6   Ada.Text_Io.Put_Line (Boolean'Val (1)'Image);
7 end Main;
```

Runtime output

```
FALSE
TRUE
```

45.2 Representation values

The representation value defines the *internal* code, used to store enumeration values in memory or CPU registers. By default, enumeration representation values are the same as the corresponding literal positions, but you can redefine them. Here, we created a copy of **Boolean** type and assigned it a custom representation.

In Ada 2022, we can get an integer value of the representation with **Type'Enum_Rep**(**Enum**) attribute:

Listing 3: main.adb

```
1  with Ada.Text_Io;
2  with Ada.Integer_Text_Io;
3
4  procedure Main is
5    type My_Boolean is new Boolean;
6    for My_Boolean use (False => 3, True => 6);
7  begin
8    Ada.Text_Io.Put ("Enum_Rep(False) =");
9    Ada.Integer_Text_Io.Put (My_Boolean'Enum_Rep (False));
10   Ada.Text_Io.New_Line;
11   Ada.Text_Io.Put ("Enum_Rep(True) =");
12   Ada.Integer_Text_Io.Put (My_Boolean'Enum_Rep (True));
13 end Main;
```

Runtime output

```
Enum_Rep(False) =
Enum_Rep(True) =
```

And, for the reverse mapping, we can use **Type'Enum_Val** (**Int**):

Listing 4: main.adb

```
1  with Ada.Text_Io;
2  with Ada.Integer_Text_Io;
3
4  procedure Main is
5    type My_Boolean is new Boolean;
6    for My_Boolean use (False => 3, True => 6);
7  begin
8    Ada.Text_Io.Put_Line (My_Boolean'Enum_Val (3)'Image);
9    Ada.Text_Io.Put_Line (My_Boolean'Enum_Val (6)'Image);
10
11   Ada.Text_Io.Put ("Pos(False) =");
12   Ada.Integer_Text_Io.Put (My_Boolean'Pos (False));
13   Ada.Text_Io.New_Line;
14   Ada.Text_Io.Put ("Pos(True) =");
15   Ada.Integer_Text_Io.Put (My_Boolean'Pos (True));
16 end Main;
```

Runtime output

```

FALSE
TRUE
Pos(False) =      0
Pos(True)  =      1

```

Note that the '`Val(X)`'/'`Pos(X)`' behaviour still is the same.

Custom representations can be useful for integration with a low level protocol or hardware.

45.3 Before Ada 2022

This doesn't initially look like an important feature, but let's see how we'd do the equivalent with Ada 2012 and earlier versions. First, we need an integer type of matching size, then we instantiate `Ada.Unchecked_Conversion`. Next, we call `To_Int/From_Int` to work with representation values. And finally an extra type conversion is needed:

Listing 5: main.adb

```

1  with Ada.Text_Io;
2  with Ada.Integer_Text_Io;
3  with Ada.Unchecked_Conversion;
4
5  procedure Main is
6
7    type My_Boolean is new Boolean;
8    for My_Boolean use (False => 3, True => 6);
9    type My_Boolean_Int is range 3 .. 6;
10   for My_Boolean_Int'Size use My_Boolean'Size;
11
12  function To_Int is new Ada.Unchecked_Conversion
13    (My_Boolean, My_Boolean_Int);
14
15  function From_Int is new Ada.Unchecked_Conversion
16    (My_Boolean_Int, My_Boolean);
17
18 begin
19  Ada.Text_Io.Put ("To_Int(False) =");
20  Ada.Integer_Text_Io.Put (Integer (To_Int (False)));
21  Ada.Text_Io.New_Line;
22  Ada.Text_Io.Put ("To_Int(True) =");
23  Ada.Integer_Text_Io.Put (Integer (To_Int (True)));
24  Ada.Text_Io.New_Line;
25  Ada.Text_Io.Put ("From_Int (3) =");
26  Ada.Text_Io.Put_Line (From_Int (3)'Image);
27  Ada.Text_Io.New_Line;
28  Ada.Text_Io.Put ("From_Int (6) =");
29  Ada.Text_Io.Put_Line (From_Int (6)'Image);
30 end Main;

```

Runtime output

```

To_Int(False) =      3
To_Int(True)  =      6
From_Int (3)  =TRUE

From_Int (6)  =TRUE

```

Even with all that, this solution doesn't work for generic formal type (because `T'Size` must be a static value)!

We should note that these new attributes may already be familiar to GNAT users because they've been in the GNAT compiler for many years.

45.4 References

- ARM 13.4 Enumeration Representation Clauses⁷⁹
- AI12-0237-1⁸⁰

⁷⁹ <http://www.adা-auth.org/standards/22aarm/html/AA-13-4.html>

⁸⁰ <http://www.adা-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0237-1.TXT>

BIG NUMBERS

Note: Big numbers are supported by

- GNAT Community Edition 2020
 - GCC 11
 - GCC 10 (draft, no user defined literals)
-

Ada 2022 introduces big integers and big real types.

46.1 Big Integers

The package `Ada.Numerics.Big_Numbers.Big_Integers` contains a type `Big_Integer` and corresponding operations such as comparison (`=, <, >, <=, >=`), arithmetic (`+, -, *, /, rem, mod, abs, **`), Min, Max and Greatest_Common_Divisor. The type also has `Integer_Literal` and `Put_Image` aspects redefined, so you can use it in a natural manner.

```
Ada.Text_Io.Put_Line (Big_Integer'Image(2 ** 256));
```

```
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

46.2 Tiny RSA implementation

Note: Note that you shouldn't use `Big_Numbers` for cryptography because it's vulnerable to timing side-channels attacks.

We can implement the RSA algorithm⁸¹ in a few lines of code. The main operation of RSA is $(m^d) \bmod n$. But you can't just write `m ** d`, because these are really big numbers and the result won't fit into memory. However, if you keep intermediate result `mod n` during the m^d calculation, it will work. Let's write this operation as a function:

Listing 1: power_mod.ads

```
1 pragma Ada_2022;
2
3 with Ada.Numerics.Big_Numbers.Big_Integers;
4 use Ada.Numerics.Big_Numbers.Big_Integers;
```

(continues on next page)

⁸¹ [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

(continued from previous page)

```

5   -- Calculate M ** D mod N
6
7
8 function Power_Mod (M, D, N : Big_Integer) return Big_Integer;

```

Listing 2: power_mod.adb

```

1  function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
2
3    function Is_Odd (X : Big_Integer) return Boolean is
4      (X mod 2 /= 0);
5
6    Result : Big_Integer := 1;
7    Exp    : Big_Integer := D;
8    Mult   : Big_Integer := M mod N;
9  begin
10   while Exp /= 0 loop
11     -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
12     if Is_Odd (Exp) then
13       Result := (Result * Mult) mod N;
14     end if;
15
16     Mult := Mult ** 2 mod N;
17     Exp := Exp / 2;
18   end loop;
19
20   return Result;
21 end Power_Mod;

```

Let's check this with the example from [Wikipedia⁸²](#). In that example, the *public key* is ($n = 3233$, $e = 17$) and the message is $m = 65$. The encrypted message is $m^e \bmod n = 65^{17} \bmod 3233 = 2790 = c$.

```
Ada.Text_Io.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
```

2790

To decrypt it with the public key ($n = 3233$, $d = 413$), we need to calculate $c^d \bmod n = 2790^{413} \bmod 3233$:

```
Ada.Text_Io.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
```

65

So 65 is the original message m . Easy!

Here is the complete code snippet:

Listing 3: main.adb

```

1  pragma Ada_2022;
2
3  with Ada.Text_Io;
4  with Ada.Numerics.Big_Numbers.Big_Integers;
5  use Ada.Numerics.Big_Numbers.Big_Integers;
6
7  procedure Main is
8

```

(continues on next page)

⁸² [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

(continued from previous page)

```

9   -- Calculate M ** D mod N
10
11  function Power_Mod (M, D, N : Big_Integer) return Big_Integer is
12
13    function Is_Odd (X : Big_Integer) return Boolean is
14      (X mod 2 /= 0);
15
16    Result : Big_Integer := 1;
17    Exp    : Big_Integer := D;
18    Mult   : Big_Integer := M mod N;
19
20  begin
21    while Exp /= 0 loop
22      -- Loop invariant is Power_Mod'Result = Result * Mult**Exp mod N
23      if Is_Odd (Exp) then
24        Result := (Result * Mult) mod N;
25      end if;
26
27      Mult := Mult ** 2 mod N;
28      Exp := Exp / 2;
29    end loop;
30
31    return Result;
32  end Power_Mod;
33
34 begin
35   Ada.Text_Io.Put_Line (Big_Integer'Image (2 ** 256));
36   -- Encrypt:
37   Ada.Text_Io.Put_Line (Power_Mod (M => 65, D => 17, N => 3233)'Image);
38   -- Decrypt:
39   Ada.Text_Io.Put_Line (Power_Mod (M => 2790, D => 413, N => 3233)'Image);
40 end Main;

```

Runtime output

```

115792089237316195423570985008687907853269984665640564039457584007913129639936
2790
65

```

46.3 Big Reals

In addition to Big_Integer, Ada 2022 provides Big Reals⁸³.

46.4 References

- ARM A.5.6 Big Integers⁸⁴
- ARM A.5.7 Big Reals⁸⁵
- AI12-0208-1⁸⁶

⁸³ <http://www.adu-auth.org/standards/22aarm/html/AA-A-5-7.html>

⁸⁴ <http://www.adu-auth.org/standards/22aarm/html/AA-A-5-6.html>

⁸⁵ <http://www.adu-auth.org/standards/22aarm/html/AA-A-5-7.html>

⁸⁶ <http://www.adu-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0208-1.TXT>

INTERFACING C VARIADIC FUNCTIONS

Note: Variadic convention is supported by

- GNAT Community Edition 2020
 - GCC 11
-

In C, [variadic functions⁸⁷](#) take a variable number of arguments and an ellipsis as the last parameter of the declaration. A typical and well-known example is:

```
int printf (const char* format, ...);
```

Usually, in Ada, we bind such a function with just the parameters we want to use:

```
procedure printf_double
  (format : Interfaces.C.char_array;
   value   : Interfaces.C.double)
  with Import,
       Convention    => C,
       External_Name => "printf";
```

Then we call it as a normal Ada function:

```
printf_double (Interfaces.C.To_C ("Pi=%f"), Ada.Numerics.pi);
```

Unfortunately, doing it this way doesn't always work because some [ABI⁸⁸](#)s use different calling conventions for variadic functions. For example, the [AMD64 ABI⁸⁹](#) specifies:

- %rax — with variable arguments passes information about the number of vector registers used;
- %xmm0–%xmm1 — used to pass and return floating point arguments.

This means, if we write (in C):

```
printf("%d", 5);
```

The compiler will place 0 into %rax, because we don't pass any float argument. But in Ada, if we write:

```
procedure printf_int
  (format : Interfaces.C.char_array;
   value   : Interfaces.C.int)
  with Import,
       Convention    => C,
```

(continues on next page)

⁸⁷ <https://en.cppreference.com/w/c/variadic>

⁸⁸ https://en.wikipedia.org/wiki/Application_binary_interface

⁸⁹ <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

(continued from previous page)

```
External_Name => "printf";
printf_int (Interfaces.C.To_C ("d=%d"), 5);
```

the compiler won't use the %rax register at all. (You can't include any float argument because there's no float parameter in the Ada wrapper function declaration.) As result, you will get a crash, stack corruption, or other undefined behavior.

To fix this, Ada 2022 provides a new family of calling convention names — C_Variadic_N:

The convention C_Variadic_n is the calling convention for a variadic C function taking n fixed parameters and then a variable number of additional parameters.

Therefore, the correct way to bind the printf function is:

```
procedure printf_int
  (format : Interfaces.C.char_array;
   value   : Interfaces.C.int)
  with Import,
       Convention    => C_Variadic_1,
       External_Name => "printf";
```

And the following call won't crash on any supported platform:

```
printf_int (Interfaces.C.To_C ("d=%d"), 5);
```

Without this convention, problems cause by this mismatch can be very hard to debug. So, this is a very useful extension to the Ada-to-C interfacing facility.

Here is the complete code snippet:

Listing 1: main.adb

```
1  with Interfaces.C;
2
3  procedure Main is
4
5    procedure printf_int
6      (format : Interfaces.C.char_array;
7       value   : Interfaces.C.int)
8    with Import,
9         Convention    => C_Variadic_1,
10        External_Name => "printf";
11
12  begin
13    printf_int (Interfaces.C.To_C ("d=%d"), 5);
14 end Main;
```

47.1 References

- ARM B.3 Interfacing with C and C++⁹⁰
- AI12-0028-1⁹¹

⁹⁰ <http://www.ada-auth.org/standards/22aarm/html/AA-B-3.html>

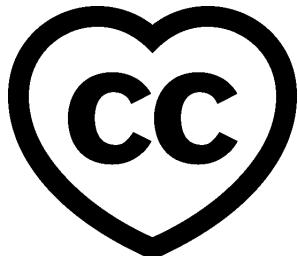
⁹¹ <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AI12s/AI12-0028-1.TXT>

Part V

Ada for the C++ or Java Developer

Copyright © 2013 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page⁹²](#)



This document will present the Ada language using terminology and examples that are familiar to developers that understand the C++ or Java languages.

This document was prepared by Quentin Ochem, with contributions and review from Richard Kenner, Albert Lee, and Ben Brosol.

⁹² <http://creativecommons.org/licenses/by-sa/4.0>

CHAPTER
FORTYEIGHT

PREFACE

Nowadays it seems like talking about programming languages is a bit passé. The technical wars of the past decade have subsided and today we see a variety of high-level and well-established languages offering functionality that can meet the needs of any programmer.

Python, Java, C++, C#, and Visual Basic are recent examples. Indeed, these languages make it easier to write code very quickly, are very flexible, offer features with highly dynamic behavior, and some even allow compilers to deduce the developer's probable intent.

Why, then, talk about yet another language? Well, by addressing the general programming market, the aforementioned languages have become poorly suited for working within the domain of high-integrity systems. In highly reliable, secure and safe applications such as those found in and around airplanes, rockets, satellites, trains, and in any device whose failure could jeopardize human life or critical assets, the programming languages used must support the high standard of software engineering necessary to maintain the integrity of the system.

The concept of verification — the practice of showing that the system behaves and performs as intended — is key in such environments. Verification can be accomplished by some combination of review, testing, static analysis, and formal proof techniques. The increasing reliance on software and increasing complexity of today's systems has made this task more difficult. Technologies and practices that might have been perfectly acceptable ten or fifteen years ago are insufficient today. Thankfully, the state of the art in analysis and proof tools and techniques has also advanced.

The latest revisions of the Ada language, Ada 2005 and Ada 2012, make enhanced software integrity possible. From its inception in the 1980s, Ada was designed to meet the requirements of high-integrity systems, and continues to be well-suited for the implementation of critical embedded or native applications. And it has been receiving increased attention recently. Every language revision has enhanced expressiveness in many areas. Ada 2012, in particular, has introduced new features for contract-based programming that are valuable to any project where verification is part of the engineering lifecycle. Along with these language enhancements, Ada compiler and tool technology has also kept pace with general computing developments over the past few years. Ada development environments are available on a wide range of platforms and are being used for the most demanding applications.

It is no secret that we at AdaCore are very enthusiastic about Ada, but we will not claim that Ada is always the solution; Ada is no more a silver bullet than any other language. In some domains other languages make sense because of the availability of particular libraries or development frameworks. For example, C++ and Java are considered good choices for desktop programs or applications where a shortened time to market is a major objective. Other areas, such as website programming or system administration, tend to rely on different formalisms such as scripting and interpreted languages. The key is to select the proper technical approach, in terms of the language and tools, to meet the requirements. Ada's strength is in areas where reliability is paramount.

Learning a new language shouldn't be complicated. Programming paradigms have not evolved much since object oriented programming gained a foothold, and the same

paradigms are present one way or another in many widely used languages. This document will thus give you an overview of the Ada language using analogies to C++ and Java — these are the languages you're already likely to know. No prior knowledge of Ada is assumed. If you are working on an Ada project now and need more background, if you are interested in learning to program in Ada, or if you need to perform an assessment of possible languages to be used for a new development, this guide is for you.

CHAPTER
FORTYNINE

BASICS

Ada implements the vast majority of programming concepts that you're accustomed to in C++ and Java: classes, inheritance, templates (generics), etc. Its syntax might seem peculiar, though. It's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a simpler language — its syntax favors making it easier to conceptualize and read program code, rather than making it faster to write in a cleverly condensed manner. For example, full words like `begin` and `end` are used in place of curly braces. Conditions are written using `if`, `then`, `elsif`, `else`, and `end if`. Ada's assignment operator does not double as an expression, smoothly eliminating any frustration that could be caused by `=` being used where `==` should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens `--` mark the start of a comment that continues to the end of the line. This is exactly the same as using `//` for comments in C++ and Java. There is no equivalent of `/* ... */` block comments in Ada; use multiple `--` lines instead.

Ada compilers are stricter with type and range checking than most C++ and Java programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding more creatively, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, dynamic checks (such as array bounds checks) provide verification that could not be done at compile time. Dynamic checks are performed at run time, similar to what is done in Java.

Ada identifiers and reserved words are case insensitive. The identifiers `VAR`, `var` and `VaR` are treated as the same; likewise `begin`, `BEGIN`, `Begin`, etc. Language-specific characters, such as accents, Greek or Russian letters, and Asian alphabets, are acceptable to use. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

abort	else	null	select
abs	elsif	of	separate
abstract	end	or	some
accept	entry	others	subtype
access	exception	out	synchronized
aliased	exit	overriding	tagged
all	for	package	task
and	function	pragma	terminate
array	generic	private	then
at	goto	procedure	type
begin	if	protected	until
body	in	raise	use
case	interface	range	when
constant	is	record	while
declare	limited	rem	with
delay	loop	renames	xor
delta	mod	requeue	
digits	new	return	
do	not	reverse	

Ada is designed to be portable. Ada compilers must follow a precisely defined international (ISO) standard language specification with clearly documented areas of vendor freedom where the behavior depends on the implementation. It's possible, then, to write an implementation-independent application in Ada and to make sure it will have the same effect across platforms and compilers.

Ada is truly a general purpose, multiple paradigm language that allows the programmer to employ or avoid features like run-time contract checking, tasking, object oriented programming, and generics. Efficiently programmed Ada is employed in device drivers, interrupt handlers, and other low-level functions. It may be found today in devices with tight limits on processing speed, memory, and power consumption. But the language is also used for programming larger interconnected systems running on workstations, servers, and supercomputers.

COMPILATION UNIT STRUCTURE

C++ programming style usually promotes the use of two distinct files: header files used to define specifications (.h*, ..hxx, .hpp), and implementation files which contain the executable code (.c, .cxx, .cpp). However, the distinction between specification and implementation is not enforced by the compiler and may need to be worked around in order to implement, for example, inlining or templates.

Java compilers expect both the implementation and specification to be in the same .java file. (Yes, design patterns allow using interfaces to separate specification from implementation to a certain extent, but this is outside of the scope of this description.)

Ada is superficially similar to the C++ case: Ada compilation units are generally split into two parts, the specification and the body. However, what goes into those files is more predictable for both the compiler and for the programmer. With GNAT, compilation units are stored in files with a .ads extension for specifications and with a .adb extension for implementations.

Without further ado, we present the famous "Hello World" in three languages:

[Ada]

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main is
begin
    Put_Line ("Hello World");
end Main;
```

[C++]

```
#include <iostream>
using namespace std;

int main(int argc, const char* argv[]) {
    cout << "Hello World" << endl;
}
```

[Java]

```
public class Main {
    public static void main(String [] argv) {
        System.out.println ("Hello World");
    }
}
```

The first line of Ada we see is the `with` clause, declaring that the unit (in this case, the Main subprogram) will require the services of the package Ada.Text_IO. This is different from how `#include` works in C++ in that it does not, in a logical sense, copy/paste the code of Ada.Text_IO into Main. The `with` clause directs the compiler to make the public

interface of the Ada.Text_Io package visible to code in the unit (here Main) containing the **with** clause. Note that this construct does not have a direct analog in Java, where the entire CLASSPATH is always accessible. Also, the name Main for the main subprogram was chosen for consistency with C++ and Java style but in Ada the name can be whatever the programmer chooses.

The **use** clause is the equivalent of **using namespace** in C++, or **import** in Java (though it wasn't necessary to use **import** in the Java example above). It allows you to omit the full package name when referring to **with**'ed units. Without the **use** clause, any reference to Ada.Text_Io items would have had to be fully qualified with the package name. The Put_Line line would then have read:

```
Ada.Text_Io.Put_Line ("Hello World");
```

The word "package" has different meanings in Ada and Java. In Java, a package is used as a namespace for classes. In Ada, it's often a compilation unit. As a result Ada tends to have many more packages than Java. Ada package specifications ("package specs" for short) have the following structure:

```
package Package_Name is
    -- public declarations
private
    -- private declarations
end Package_Name;
```

The implementation in a package body (written in a .adb file) has the structure:

```
package body Package_Name is
    -- implementation
end Package_Name;
```

The **private** reserved word is used to mark the start of the private portion of a package spec. By splitting the package spec into private and public parts, it is possible to make an entity available for use while hiding its implementation. For instance, a common use is declaring a **record** (Ada's **struct**) whose fields are only visible to its package and not to the caller. This allows the caller to refer to objects of that type, but not to change any of its contents directly.

The package body contains implementation code, and is only accessible to outside code through declarations in the package spec.

An entity declared in the private part of a package in Ada is roughly equivalent to a protected member of a C++ or Java class. An entity declared in the body of an Ada package is roughly equivalent to a private member of a C++ or Java class.

STATEMENTS, DECLARATIONS, AND CONTROL STRUCTURES

51.1 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[Ada]

```
--  
-- Ada program to declare and modify Integers  
--  
procedure Main is  
    -- Variable declarations  
    A, B : Integer := 0;  
    C      : Integer := 100;  
    D      : Integer;  
begin  
    -- Ada uses a regular assignment statement for incrementation.  
    A := A + 1;  
  
    -- Regular addition  
    D := A + B + C;  
end Main;
```

[C++]

```
/*  
 * C++ program to declare and modify ints  
 */  
int main(int argc, const char* argv[]) {  
    // Variable declarations  
    int a = 0, b = 0, c = 100, d;  
  
    // C++ shorthand for incrementation  
    a++;  
  
    // Regular addition  
    d = a + b + c;  
}
```

[Java]

```
/*  
 * Java program to declare and modify ints  
 */  
public class Main {
```

(continues on next page)

(continued from previous page)

```
public static void main(String [] argv) {
    // Variable declarations
    int a = 0, b = 0, c = 100, d;

    // Java shorthand for incrementation
    a++;

    // Regular addition
    d = a + b + c;
}
```

Statements are terminated by semicolons in all three languages. In Ada, blocks of code are surrounded by the reserved words **begin** and **end** rather than by curly braces. We can use both multi-line and single-line comment styles in the C++ and Java code, and only single-line comments in the Ada code.

Ada requires variable declarations to be made in a specific area called the *declarative part*, seen here before the **begin** keyword. Variable declarations start with the identifier in Ada, as opposed to starting with the type as in C++ and Java (also note Ada's use of the `:` separator). Specifying initializers is different as well: in Ada an initialization expression can apply to multiple variables (but will be evaluated separately for each), whereas in C++ and Java each variable is initialized individually. In all three languages, if you use a function as an initializer and that function returns different values on every invocation, each variable will get initialized to a different value.

Let's move on to the imperative statements. Ada does not provide `++` or `--` shorthand expressions for increment/decrement operations; it is necessary to use a full assignment statement. The `:=` symbol is used in Ada to perform value assignment. Unlike C++'s and Java's `=` symbol, `:=` can not be used as part of an expression. So, a statement like `A := B := C;` doesn't make sense to an Ada compiler, and neither does a clause like `if A := B then` Both are compile-time errors.

You can nest a block of code within an outer block if you want to create an inner scope:

```
with Ada.Text_Io; use Ada.Text_Io;

procedure Main is
begin
    Put_Line ("Before the inner block");

    declare
        Alpha : Integer := 0;
    begin
        Alpha := Alpha + 1;
        Put_Line ("Now inside the inner block");
    end;

    Put_Line ("After the inner block");
end Main;
```

It is OK to have an empty declarative part or to omit the declarative part entirely — just start the inner block with **begin** if you have no declarations to make. However it is not OK to have an empty sequence of statements. You must at least provide a **null**; statement, which does nothing and indicates that the omission of statements is intentional.

51.2 Conditions

The use of the **if** statement:

[Ada]

```
if Variable > 0 then
    Put_Line (" > 0 ");
elsif Variable < 0 then
    Put_Line (" < 0 ");
else
    Put_Line (" = 0 ");
end if;
```

[C++]

```
if (Variable > 0)
    cout << " > 0 " << endl;
else if (Variable < 0)
    cout << " < 0 " << endl;
else
    cout << " = 0 " << endl;
```

[Java]

```
if (Variable > 0)
    System.out.println (" > 0 ");
else if (Variable < 0)
    System.out.println (" < 0 ");
else
    System.out.println (" = 0 ");
```

In Ada, everything that appears between the **if** and **then** keywords is the conditional expression — no parentheses required. Comparison operators are the same, except for equality (=) and inequality (/=). The English words **not**, **and**, and **or** replace the symbols !, &, and |, respectively, for performing boolean operations.

It's more customary to use **&&** and **||** in C++ and Java than **&** and **|** when writing boolean expressions. The difference is that **&&** and **||** are short-circuit operators, which evaluate terms only as necessary, and **&** and **|** will unconditionally evaluate all terms. In Ada, **and** and **or** will evaluate all terms; **and then** and **or else** direct the compiler to employ short circuit evaluation.

Here are what switch/case statements look like:

[Ada]

```
case Variable is
    when 0 =>
        Put_Line ("Zero");
    when 1 .. 9 =>
        Put_Line ("Positive Digit");
    when 10 | 12 | 14 | 16 | 18 =>
        Put_Line ("Even Number between 10 and 18");
    when others =>
        Put_Line ("Something else");
end case;
```

[C++]

```
switch (Variable) {
    case 0:
```

(continues on next page)

(continued from previous page)

```

cout << "Zero" << endl;
break;
case 1: case 2: case 3: case 4: case 5:
case 6: case 7: case 8: case 9:
    cout << "Positive Digit" << endl;
    break;
case 10: case 12: case 14: case 16: case 18:
    cout << "Even Number between 10 and 18" << endl;
    break;
default:
    cout << "Something else";
}

```

[Java]

```

switch (Variable) {
    case 0:
        System.out.println ("Zero");
        break;
    case 1: case 2: case 3: case 4: case 5:
    case 6: case 7: case 8: case 9:
        System.out.println ("Positive Digit");
        break;
    case 10: case 12: case 14: case 16: case 18:
        System.out.println ("Even Number between 10 and 18");
        break;
    default:
        System.out.println ("Something else");
}

```

In Ada, the `case` and `end case` lines surround the whole case statement, and each case starts with `when`. So, when programming in Ada, replace `switch` with `case`, and replace `case` with `when`.

Case statements in Ada require the use of discrete types (integers or enumeration types), and require all possible cases to be covered by `when` statements. If not all the cases are handled, or if duplicate cases exist, the program will not compile. The default case, `default:` in C++ and Java, can be specified using `when others =>` in Ada.

In Ada, the `break` instruction is implicit and program execution will never fall through to subsequent cases. In order to combine cases, you can specify ranges using `..` and enumerate disjoint values using `|` which neatly replaces the multiple `case` statements seen in the C++ and Java versions.

51.3 Loops

In Ada, loops always start with the `loop` reserved word and end with `end loop`. To leave the loop, use `exit` — the C++ and Java equivalent being `break`. This statement can specify a terminating condition using the `exit when` syntax. The `loop` opening the block can be preceded by a `while` or a `for`.

The `while` loop is the simplest one, and is very similar across all three languages:

[Ada]

```

while Variable < 10_000 loop
    Variable := Variable * 2;
end loop;

```

[C++]

```
while (Variable < 10000) {
    Variable = Variable * 2;
}
```

[Java]

```
while (Variable < 10000) {
    Variable = Variable * 2;
}
```

Ada's **for** loop, however, is quite different from that in C++ and Java. It always increments or decrements a loop index within a discrete range. The loop index (or "loop parameter" in Ada parlance) is local to the scope of the loop and is implicitly incremented or decremented at each iteration of the loop statements; the program cannot directly modify its value. The type of the loop parameter is derived from the range. The range is always given in ascending order even if the loop iterates in descending order. If the starting bound is greater than the ending bound, the interval is considered to be empty and the loop contents will not be executed. To specify a loop iteration in decreasing order, use the **reverse** reserved word. Here are examples of loops going in both directions:

[Ada]

```
-- Outputs 0, 1, 2, ..., 9
for Variable in 0 .. 9 loop
    Put_Line (Integer'Image (Variable));
end loop;

-- Outputs 9, 8, 7, ..., 0
for Variable in reverse 0 .. 9 loop
    Put_Line (Integer'Image (Variable));
end loop;
```

[C++]

```
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
    cout << Variable << endl;
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
    cout << Variable << endl;
}
```

[Java]

```
// Outputs 0, 1, 2, ..., 9
for (int Variable = 0; Variable <= 9; Variable++) {
    System.out.println (Variable);
}

// Outputs 9, 8, 7, ..., 0
for (int Variable = 9; Variable >= 0; Variable--) {
    System.out.println (Variable);
}
```

Ada uses the **Integer** type's '**Image**' attribute to convert a numerical value to a String. There is no implicit conversion between **Integer** and **String** as there is in C++ and Java. We'll have a more in-depth look at such attributes later on.

It's easy to express iteration over the contents of a container (for instance, an array, a list,

or a map) in Ada and Java. For example, assuming that Int_List is defined as an array of Integer values, you can use:

[Ada]

```
for I of Int_List loop
    Put_Line (Integer'Image (I));
end loop;
```

[Java]

```
for (int i : Int_List) {
    System.out.println (i);
}
```

TYPE SYSTEM

52.1 Strong Typing

One of the main characteristics of Ada is its strong typing (i.e., relative absence of implicit type conversions). This may take some getting used to. For example, you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

```
procedure Strong_Typing is
    Alpha : Integer := 1;
    Beta  : Integer := 10;
    Result : Float;
begin
    Result := Float(Alpha) / Float(Beta);
end Strong_Typing;
```

[C++]

```
void weakTyping () {
    int alpha = 1;
    int beta = 10;
    float result;

    result = alpha / beta;
}
```

[Java]

```
void weakTyping () {
    int alpha = 1;
    int beta = 10;
    float result;

    result = alpha / beta;
}
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from Integer to Float explicit. In fact it significantly changes the behavior of the computation. While the Ada code performs a floating point operation $1.0 / 10.0$ and stores 0.1 in Result, the C++ and Java versions instead store 0.0 in result. This is because the C++ and Java versions perform an integer operation between two integer variables: $1 / 10$ is 0. The result of the integer division is then converted to a float and stored. Errors of this sort can be very hard to locate in complex

pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to **Float**:

```
-- Perform an Integer division then convert to Float  
Result := Float (Alpha / Beta);
```

In Ada, a floating point literal must be written with both an integral and decimal part. **10** is not a valid literal for a floating point value, while **10.0** is.

52.2 Language-Defined Types

The principal scalar types predefined by Ada are **Integer**, **Float**, **Boolean**, and **Character**. These correspond to **int**, **float**, **bool/boolean**, and **char**, respectively. The names for these types are not reserved words; they are regular identifiers.

52.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators **+**, **-**, *****, and **/** for integer and floating-point types take operands of the same type and return a value of that type.

```
procedure Main is  
    type Distance is new Float;  
    type Area is new Float;  
  
    D1 : Distance := 2.0;  
    D2 : Distance := 3.0;  
    A : Area;  
begin  
    D1 := D1 + D2;          -- OK  
    D1 := D1 + A;           -- NOT OK: incompatible types for "+" operator  
    A := D1 * D2;           -- NOT OK: incompatible types for ":=" assignment  
    A := Area (D1 * D2);   -- OK  
end Main;
```

Even though the **Distance** and **Area** types above are just **Floats**, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two **Distances** yields a **Distance**) and prohibit some useful cases (for example multiplying two **Distances** should deliver an **Area**). These situations can be handled through other mechanisms. A predefined operation can be identified as **abstract** to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its

operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C++ and Java's **enums**.

[Ada]

```
type Day is
  (Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday);
```

[C++]

```
enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday};
```

[Java]

```
enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday}
```

But even though such enumerations may be implemented using a machine word, at the language level Ada will not confuse the fact that Monday is a Day and is not an **Integer**. You can compare a Day with another Day, though. To specify implementation details like the numeric values that correspond with enumeration values in C++ you include them in the original **enum** statement:

[C++]

```
enum Day {
  Monday    = 10,
  Tuesday   = 11,
  Wednesday = 12,
  Thursday  = 13,
  Friday    = 14,
  Saturday  = 15,
  Sunday    = 16};
```

But in Ada you must use both a type definition for Day as well as a separate *representation clause* for it like:

[Ada]

```
for Day use
  (Monday    => 10,
   Tuesday   => 11,
```

(continues on next page)

(continued from previous page)

```
Wednesday => 12,
Thursday  => 13,
Friday    => 14,
Saturday  => 15,
Sunday    => 16);
```

52.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a *range constraint* introduced with the `range` reserved word, for example:

```
procedure Main is
    type Grade is range 0 .. 100;

    G1, G2 : Grade;
    N      : Integer;
begin
    ...           -- Initialization of N
    G1 := 80;     -- OK
    G1 := N;      -- Illegal (type mismatch)
    G1 := Grade (N); -- Legal, run-time range check
    G2 := G1 + 10; -- Legal, run-time range check
    G1 := (G1 + G2)/2; -- Legal, run-time range check
end Main;
```

In the above example, `Grade` is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to `G1` is correct and will not raise a run-time exception. Assigning `N` to `G1` is illegal since `Grade` is a different type than `Integer`. Converting `N` to `Grade` makes the assignment legal, and a range check on the conversion confirms that the value is within `0 .. 100`. Assigning `G1+10` to `G2` is legal since `+` for `Grade` returns a `Grade` (note that the literal `10` is interpreted as a `Grade` value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression `G1 + G2` may be outside the range of `Grade`, but the final result will be in range. Nevertheless, depending on the representation chosen for `Grade`, the addition may overflow. If the compiler represents `Grade` values as signed 8-bit integers (i.e., machine numbers in the range `-128 .. 127`) then the sum `G1+G2` may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

```
G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
```

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the `Day` example, one can write:

```
type Business_Day is new Day range Monday .. Friday;
type Weekend_Day is new Day range Saturday .. Sunday;
```

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind *subtypes* in Ada. A subtype is a type with optional additional constraints. For example:

```
subtype Business_Day is Day range Monday .. Friday;
subtype Weekend_Day is Day range Saturday .. Sunday;
subtype Dice_Throw is Integer range 1 .. 6;
```

These declarations don't create new types, just new names for constrained ranges of their base types.

52.5 Generalized Type Contracts: Subtype Predicates

Range checks are a special form of type contracts; a more general method is provided by Ada subtype predicates, introduced in Ada 2012. A subtype predicate is a boolean expression defining conditions that are required for a given type or subtype. For example, the Dice_Throw subtype shown above can be defined in the following way:

```
subtype Dice_Throw is Integer
  with Dynamic_Predicate => Dice_Throw in 1 .. 6;
```

The clause beginning with **with** introduces an Ada aspect, which is additional information provided for declared entities such as types and subtypes. The **Dynamic_Predicate** aspect is the most general form. Within the predicate expression, the name of the (sub)type refers to the current value of the (sub)type. The predicate is checked on assignment, parameter passing, and in several other contexts. There is a **Static_Predicate** form which introduce some optimization and constrains on the form of these predicates, outside of the scope of this document.

Of course, predicates are useful beyond just expressing ranges. They can be used to represent types with arbitrary constraints, in particular types with discontinuities, for example:

```
type Not_Null is new Integer
  with Dynamic_Predicate => Not_Null /= 0;

type Even is new Integer
  with Dynamic_Predicate => Even mod 2 = 0;
```

52.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The '**Image**' and '**Value**' attributes allow you to transform a scalar value into a **String** and vice-versa. For example:

```
declare
  A : Integer := 99;
begin
  Put_Line (Integer'Image (A));
```

(continues on next page)

(continued from previous page)

```
A := Integer'Value ("99");
end;
```

Certain attributes are provided only for certain kinds of types. For example, the '`Val`' and '`Pos`' attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

```
declare
  C : Character := 'a';
begin
  C := Character'Val (Character'Pos (C) + 1);
end;
```

A more concise way to get the next value in Ada is to use the '`Succ`' attribute:

```
declare
  C : Character := 'a';
begin
  C := Character'Succ (C);
end;
```

You can get the previous value using the '`Pred`' attribute. Here is the equivalent in C++ and Java:

[C++]

```
char c = 'a';
c++;
```

[Java]

```
char c = 'a';
c++;
```

Other interesting examples are the '`First`' and '`Last`' attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, `Integer'First` returns -2^{31} and `Integer'Last` returns $2^{31} - 1$.

52.7 Arrays and Strings

C++ arrays are pointers with offsets, but the same is not the case for Ada and Java. Arrays in the latter two languages are not interchangeable with operations on pointers, and array types are considered first-class citizens. Arrays in Ada have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from '`a`' to '`z`':

[Ada]

```
declare
  type Arr_Type is array (Integer range <>) of Character;
```

(continues on next page)

(continued from previous page)

```

Arr : Arr_Type (1 .. 26);
C : Character := 'a';
begin
  for I in Arr'Range loop
    Arr (I) := C;
    C := Character'Succ (C);
  end loop;
end;

```

[C++]

```

char Arr [26];
char C = 'a';

for (int I = 0; I < 26; ++I) {
  Arr [I] = C;
  C = C + 1;
}

```

[Java]

```

char [] Arr = new char [26];
char C = 'a';

for (int I = 0; I < Arr.length; ++I) {
  Arr [I] = C;
  C = C + 1;
}

```

In C++ and Java, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of Integer as the index type, but does not provide any constraints (use `<>`, pronounced *box*, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called '`Range`'. In our example, `Arr'Range` can also be expressed as `Arr'First .. Arr'Last`; both expressions will resolve to `1 .. 26`. So the '`Range`' attribute supplies the bounds for our `for` loop. There is no risk of stating either of the bounds incorrectly, as one might do in C++ where `I <= 26` may be specified as the end-of-loop condition.

As in C++, Ada **Strings** are arrays of **Characters**. The C++ or Java String class is the equivalent of the Ada type `Ada.Strings.Unbounded_String` which offers additional capabilities in exchange for some overhead. Ada strings, importantly, are not delimited with the special character '`'0'`' like they are in C++. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined **String** type is very straightforward to use:

```
My_String : String (1 .. 26);
```

Unlike C++ and Java, Ada does not offer escape sequences such as '`'n'`'. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, `&`). Here for example, is how to initialize a line of text ending with a new line:

```
My_String : String := "This is a line with a end of line" & ASCII.LF;
```

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of `My_String`'s bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C++ or Java, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada,

Learning Ada

the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

[Ada]

```
declare
    type Arr_Type is array (Integer range <>) of Integer;
    A1 : Arr_Type (1 .. 2);
    A2 : Arr_Type (1 .. 2);
begin
    A1 (1) := 0;
    A1 (2) := 1;

    A2 := A1;
end;
```

[C++]

```
int A1 [2];
int A2 [2];

A1 [0] = 0;
A1 [1] = 1;

for (int i = 0; i < 2; ++i) {
    A2 [i] = A1 [i];
}
```

[Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];

A1 [0] = 0;
A1 [1] = 1;

A2 = Arrays.copyOf(A1, A1.length);
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

```
declare
    type Arr_Type is array (Integer range <>) of Integer
    A1 : Arr_Type (1 .. 10);
    A2 : Arr_Type (1 .. 5);
begin
    A2 (1 .. 3) := A1 (4 .. 6);
end;
```

This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2. Note that only the length matters here: the values of the indexes don't have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

[Ada]

```
declare
    type Arr_Type is array (Integer range <>) of Integer;
```

(continues on next page)

(continued from previous page)

```
A1 : Arr_Type (1 .. 2);
A2 : Arr_Type (1 .. 2);
begin
  if A1 = A2 then
```

[C++]

```
int A1 [2];
int A2 [2];

bool eq = true;

for (int i = 0; i < 2; ++i) {
  if (A1 [i] != A2 [i]) {
    eq = false;
  }
}

if (eq) {
```

[]Java]

```
int [] A1 = new int [2];
int [] A2 = new int [2];

if (Arrays.equals (A1, A2)) {
```

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the **others** expression to define a default value for all the unspecified array elements. Therefore, you can write:

```
declare
  type Arr_Type is array (Integer range <>) of Integer;
  A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
  A2 : Arr_Type (-2 .. 42) := (others => 0);
begin
  A1 := (1, 2, 3, others => 10);

  -- use a slice to assign A2 elements 11 .. 19 to 1
  A2 (11 .. 19) := (others => 1);
end;
```

52.8 Heterogeneous Data Structures

In Ada, there's no distinction between **struct** and **class** as there is in C++. All heterogeneous data structures are **records**. Here are some simple records:

[Ada]

```
declare
  type R is record
    A, B : Integer;
    C      : Float;
  end record;

  V : R;
```

(continues on next page)

(continued from previous page)

```
begin
  V.A := 0;
end;
```

[C++]

```
struct R {
  int A, B;
  float C;
};

R V;
V.A = 0;
```

[Java]

```
class R {
  public int A, B;
  public float C;
}

R V = new R ();
V.A = 0;
```

Ada allows specification of default values for fields just like C++ and Java. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by **others** => <> to specify that fields not listed will take their default values. For example:

```
type R is record
  A, B : Integer := 0;
  C     : Float := 0.0;
end record;

V1 : R := (1, 2, 1.0);
V2 : R := (A => 1, B => 2, C => 1.0);
V3 : R := (C => 1.0, A => 1, B => 2);
V4 : R := (C => 1.0, others => <>);
```

52.9 Pointers

Pointers, references, and access types differ in significant ways across the languages that we are examining. In C++, pointers are integral to a basic understanding of the language, from array manipulation to proper declaration and use of function parameters. In Java, direct pointer manipulation is abstracted by the Java runtime. And in Ada, direct pointer manipulation is possible, but unlike C++, they are not required for basic usage with arrays and parameter passing.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

```
declare
  type R is record
    A, B : Integer;
  end record;
```

(continues on next page)

(continued from previous page)

```
V1, V2 : R;
begin
  V1.A := 0;
  V2 := V1;
  V2.A := 1;
end;
```

[C++]

```
struct R {
  int A, B;
};

R V1, V2;
V1.A = 0;
V2 = V1;
V2.A = 1;
```

[Java]

```
public class R {
  public int A, B;
}

R V1, V2;
V1 = new R ();
V1.A = 0;
V2 = V1;
V2.A = 1;
```

There's a fundamental difference between the Ada and C++ semantics above and the semantics for Java. In Ada and C++, objects are allocated on the stack and are directly accessed. V1 and V2 are two different objects and the assignment statement copies the value of V1 into V2. In Java, V1 and V2 are two *references* to objects of class R. Note that when V1 and V2 are declared, no actual object of class R yet exists in memory: it has to be allocated later with the `new` allocator operator. After the assignment `V2 = V1`, there's only one R object in memory: the assignment is a reference assignment, not a value assignment. At the end of the Java code, V1 and V2 are two references to the same objects and the `V2.A = 1` statement changes the field of that one object, while in the Ada and the C++ case V1 and V2 are two distinct objects.

To obtain similar behavior in Ada, you can use pointers. It can be done through Ada's *access type*:

[Ada]

```
declare
  type R is record
    A, B : Integer;
  end record;
  type R_Access is access R;

  V1 : R_Access;
  V2 : R_Access;
begin
  V1 := new R;
  V1.A := 0;
  V2 := V1;
  V2.A := 1;
end;
```

[C++]

```
struct R {
    int A, B;
};

R * V1, * V2;
V1 = new R ();
V1->A = 0;
V2 = V1;
V2->A = 0;
```

For those coming from the Java world: there's no garbage collector in Ada, so objects allocated by the `new` operator need to be expressly freed.

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append `.all`. The equivalent of `V1->A` in C++ can be written either as `V1.A` or `V1.all.A`.

Pointers to scalar objects in Ada and C++ look like:

[Ada]

```
procedure Main is
    type A_Int is access Integer;
    Var : A_Int := new Integer;
begin
    Var.all := 0;
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
    int * Var = new int;
    *Var = 0;
}
```

An initializer can be specified with the allocation by appending '(value):

```
Var : A_Int := new Integer'(0);
```

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being `aliased`. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either `access all` (if the referenced object needs to be assigned to) or `access constant` (if the referenced object is a constant). The '`Access`' attribute works like the C++ `&` operator to get a pointer to the object, but with a "scope accessibility" check to prevent references to objects that have gone out of scope. For example:

[Ada]

```
type A_Int is access all Integer;
Var : aliased Integer;
Ptr : A_Int := Var'Access;
```

[C++]

```
int Var;
int * Ptr = &Var;
```

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs — it's called `Ada.Unchecked_Deallocation`. To create your customized

deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

```
with Ada.Unchecked_Deallocation;
procedure Main is
    type Integer_Access is access all Integer;
    procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);
    My_Pointer : Integer_Access := new Integer;
begin
    Free (My_Pointer);
end Main;
```

[C++]

```
int main (int argc, char *argv[]) {
    int * my_pointer = new int;
    delete my_pointer;
}
```


FUNCTIONS AND PROCEDURES

53.1 General Form

Subroutines in C++ and Java are always expressed as functions (methods) which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term "subprogram" to refer to both functions and procedures.

Parameters can be passed in three distinct modes: **in**, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram. **out** is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller. **in out** is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant reference in C++). Ada also provides **access** parameters, in effect an explicit pass-by-reference indicator.

In Ada the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). (There are some exceptions to the "in general". For example, parameters of scalar types are always passed by copy, for all three modes.) C++ has the programmer specify how to pass the parameter, and Java forces primitive type parameters to be passed by copy and all other parameters to be passed by reference. For this reason, a 1:1 mapping between Ada and Java isn't obvious but here's an attempt to show these differences:

[Ada]

```
procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer);

function Func (Var : Integer) return Integer;

procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer)
is
begin
  Var2 := Func (Var1);
  Var3 := Var3 + 1;
end Proc;

function Func (Var : Integer) return Integer
is
begin
  return Var + 1;
end Func;
```

[C++]

```
void Proc
  (int Var1,
   int & Var2,
   int & Var3);

int Func (int Var);

void Proc
  (int Var1,
   int & Var2,
   int & Var3) {

  Var2 = Func (Var1);
  Var3 = Var3 + 1;
}

int Func (int Var) {
  return Var + 1;
}
```

[Java]

```
public class ProcData {
  public int Var2;
  public int Var3;

  public void Proc (int Var1) {
    Var2 = Func (Var1);
    Var3 = Var3 + 1;
  }

  public static int Func (int Var) {
    return Var + 1;
  }
}
```

The first two declarations for Proc and Func are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C++, a function that has not yet been seen cannot be used. Here, Proc can call Func because its specification has been declared. In Java, it's fine to have the declaration of the subprogram later .

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax, including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

53.2 Overloading

Different subprograms may share the same name; this is called "overloading." As long as the subprogram signatures (subprogram name, parameter types, and return types) are different, the compiler will be able to resolve the calls to the proper destinations. For example:

```
function Value (Str : String) return Integer;
function Value (Str : String) return Float;

V : Integer := Value ("8");
```

The Ada compiler knows that an assignment to V requires an **Integer**. So, it chooses the Value function that returns an **Integer** to satisfy this requirement.

Operators in Ada can be treated as functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To express an operator as a function, enclose it in quotes:

[Ada]

```
function "=" (Left : Day; Right : Integer) return Boolean;
```

[C++]

```
bool operator = (Day Left, int Right);
```

53.3 Subprogram Contracts

You can express the expected inputs and outputs of subprograms by specifying subprogram contracts. The compiler can then check for valid conditions to exist when a subprogram is called and can check that the return value makes sense. Ada allows defining contracts in the form of Pre and Post conditions; this facility was introduced in Ada 2012. They look like:

```
function Divide (Left, Right : Float) return Float
  with Pre => Right /= 0.0,
        Post => Divide'Result * Right < Left + 0.0001
              and then Divide'Result * Right > Left - 0.0001;
```

The above example adds a Pre condition, stating that Right cannot be equal to 0.0. While the IEEE floating point standard permits divide-by-zero, you may have determined that use of the result could still lead to issues in a particular application. Writing a contract helps to detect this as early as possible. This declaration also provides a Post condition on the result.

Postconditions can also be expressed relative to the value of the input:

```
procedure Increment (V : in out Integer)
  with Pre => V < Integer'Last,
        Post => V = V'Old + 1;
```

V'Old in the postcondition represents the value that V had before entering Increment.

CHAPTER
FIFTYFOUR

PACKAGES

54.1 Declaration Protection

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C++. An Ada package contains three parts that, for GNAT, are separated into two files: .ads files contain public and private Ada specifications, and .adb files contain the implementation, or Ada bodies.

Java doesn't provide any means to cleanly separate the specification of methods from their implementation: they all appear in the same file. You can use interfaces to emulate having separate specifications, but this requires the use of OOP techniques which is not always practical.

Ada and C++ do offer separation between specifications and implementations out of the box, independent of OOP.

```
package Package_Name is
    -- public specifications
private
    -- private specifications
end Package_Name;

package body Package_Name is
    -- implementation
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. The **private** keyword splits the package spec into "public" and "private" parts. That is somewhat analogous to C++'s partitioning of the class construct into different sections with different visibility properties. In Java, the encapsulation has to be done field by field, but in Ada the entire definition of a type can be hidden. For example:

```
package Types is
    type Type_1 is private;
    type Type_2 is private;
    type Type_3 is private;
    procedure P (X : Type_1);
    ...
private
    procedure Q (Y : Type_1);
    type Type_1 is new Integer range 1 .. 1000;
    type Type_2 is array (Integer range 1 .. 1000) of Integer;
    type Type_3 is record
        A, B : Integer;
    end record;
end Types;
```

Subprograms declared above the **private** separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implementation, has access to both parts.

54.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

```
-- root-child.ads

package Root.Child is
    -- package spec goes here
end Root.Child;

-- root-child.adb

package body Root.Child is
    -- package body goes here
end Root.Child;
```

Here, Root.Child is a child package of Root. The public part of Root.Child has access to the public part of Root. The private part of Child has access to the private part of Root, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

54.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a **with** clause that references the package, which is similar to the C++ **#include** directive. Visibility is implicit in Java: you can always access all classes located in your *CLASSPATH*. After a **with** clause, entities needs to be prefixed by the name of their package, like a C++ namespace or a Java package. This prefix can be omitted if a **use** clause is employed, similar to a C++ **using namespace** or a Java **import**.

[Ada]

```
-- pck.ads

package Pck is
    My_Glob : Integer;
end Pck;

-- main.adb

with Pck;

procedure Main is
begin
    Pck.My_Glob := 0;
end Main;
```

[C++]

```
// pck.h

namespace pck {
    extern int myGlob;
}

// pck.cpp

namespace pck {
    int myGlob;
}

// main.cpp

#include "pck.h"

int main (int argc, char ** argv) {
    pck::myGlob = 0;
}
```

[Java]

```
// Globals.java

package pck;

public class Globals {
    public static int myGlob;
}

// Main.java

public class Main {
    public static void main (String [] argv) {
        pck.Globals.myGlob = 0;
    }
}
```


CLASSES AND OBJECT ORIENTED PROGRAMMING

55.1 Primitive Subprograms

Primitive subprograms in Ada are basically the subprograms that are eligible for inheritance / derivation. They are the equivalent of C++ member functions and Java instance methods. While in C++ and Java these subprograms are located within the nested scope of the type, in Ada they are simply declared in the same scope as the type. There's no syntactic indication that a subprogram is a primitive of a type.

The way to determine whether P is a primitive of a type T is if

1. it is declared in the same scope as T, and
2. it contains at least one parameter of type T, or returns a result of type T.

In C++ or Java, the self reference `this` is implicitly declared. It may need to be explicitly stated in certain situations, but usually it's omitted. In Ada the self-reference, called the *controlling parameter*, must be explicitly specified in the subprogram parameter list. While it can be any parameter in the profile with any name, we'll focus on the typical case where the first parameter is used as the self parameter. Having the controlling parameter listed first also enables the use of OOP prefix notation which is convenient.

A `class` in C++ or Java corresponds to a `tagged type` in Ada. Here's an example of the declaration of an Ada tagged type with two parameters and some dispatching and non-dispatching primitives, with equivalent examples in C++ and Java:

[Ada]

```
type T is tagged record
    V, W : Integer;
end record;

type T_Access is access all T;

function F (V : T) return Integer;

procedure P1 (V : access T);

procedure P2 (V : T_Access);
```

[C++]

```
class T {
public:
    int V, W;

    int F ();
    void P1 ();
};
```

(continues on next page)

(continued from previous page)

```
};  
void P2 (T * v);
```

[Java]

```
public class T {  
    public int V, W;  
  
    public int F () {};  
  
    public void P1 () {};  
  
    public static void P2 (T v) {};  
}
```

Note that P2 is not a primitive of T — it does not have any parameters of type T. Its parameter is of type T_Access, which is a different type.

Once declared, primitives can be called like any subprogram with every necessary parameter specified, or called using prefix notation. For example:

[Ada]

```
declare  
    V : T;  
begin  
    V.P1;  
end;
```

[C++]

```
{  
    T v;  
    v.P1 ();  
}
```

[Java]

```
{  
    T v = new T ();  
    v.P1 ();  
}
```

55.2 Derivation and Dynamic Dispatch

Despite the syntactic differences, derivation in Ada is similar to derivation (inheritance) in C++ or Java. For example, here is a type hierarchy where a child class overrides a method and adds a new method:

[Ada]

```
type Root is tagged record  
    F1 : Integer;  
end record;  
  
procedure Method_1 (Self : Root);
```

(continues on next page)

(continued from previous page)

```

type Child is new Root with record
  F2 : Integer;
end record;

overriding
procedure Method_1 (Self : Child);

procedure Method_2 (Self : Child);

```

[C++]

```

class Root {
  public:
    int f1;
    virtual void method1 ();
};

class Child : public Root {
  public:
    int f2;
    virtual void method1 ();
    virtual void method2 ();
};

```

[Java]

```

public class Root {
  public int f1;
  public void method1 ();
}

public class Child extends Root {
  public int f2;
  @Override
  public void method1 ();
  public void method2 ();
}

```

Like Java, Ada primitives on tagged types are always subject to dispatching; there is no need to mark them **virtual**. Also like Java, there's an optional keyword **overriding** to ensure that a method is indeed overriding something from the parent type.

Unlike many other OOP languages, Ada differentiates between a reference to a specific tagged type, and a reference to an entire tagged type hierarchy. While Root is used to mean a specific type, Root'Class — a class-wide type — refers to either that type or any of its descendants. A method using a parameter of such a type cannot be overridden, and must be passed a parameter whose type is of any of Root's descendants (including Root itself).

Next, we'll take a look at how each language finds the appropriate method to call within an OO class hierarchy; that is, their dispatching rules. In Java, calls to non-private instance methods are always dispatching. The only case where static selection of an instance method is possible is when calling from a method to the **super** version.

In C++, by default, calls to virtual methods are always dispatching. One common mistake is to use a by-copy parameter hoping that dispatching will reach the real object. For example:

```

void proc (Root p) {
  p.method1 ();
}

```

(continues on next page)

(continued from previous page)

```
Root * v = new Child ();  
proc (*v);
```

In the above code, `p.method1()` will not dispatch. The call to `proc` makes a copy of the `Root` part of `v`, so inside `proc`, `p.method1()` refers to the `method1()` of the root object. The intended behavior may be specified by using a reference instead of a copy:

```
void proc (Root & p) {  
    p.method1 ();  
}  
  
Root * v = new Child ();  
  
proc (*v);
```

In Ada, tagged types are always passed by reference but dispatching only occurs on class-wide types. The following Ada code is equivalent to the latter C++ example:

```
declare  
procedure Proc (P : Root'Class) is  
begin  
    P.Method_1;  
end;  
  
type Root_Access is access all Root'Class;  
V : Root_Access := new Child;  
begin  
    Proc (V.all);  
end;
```

Dispatching from within primitives can get tricky. Let's consider a call to `Method_1` in the implementation of `Method_2`. The first implementation that might come to mind is:

```
procedure Method_2 (P : Root) is  
begin  
    P.Method_1;  
end;
```

However, `Method_2` is called with a parameter that is of the definite type `Root`. More precisely, it is a definite view of a child. So, this call is not dispatching; it will always call `Method_1` of `Root` even if the object passed is a child of `Root`. To fix this, a view conversion is necessary:

```
procedure Method_2 (P : Root) is  
begin  
    Root'Class (P).Method_1;  
end;
```

This is called "redispatching." Be careful, because this is the most common mistake made in Ada when using OOP. In addition, it's possible to convert from a class wide view to a definite view, and to select a given primitive, like in C++:

[Ada]

```
procedure Proc (P : Root'Class) is  
begin  
    Root (P).Method_1;  
end;
```

[C++]

```
void proc (Root & p) {
    p.Root::method1 ();
}
```

55.3 Constructors and Destructors

Ada does not have constructors and destructors in quite the same way as C++ and Java, but there is analogous functionality in Ada in the form of default initialization and finalization.

Default initialization may be specified for a record component and will occur if a variable of the record type is not assigned a value at initialization. For example:

```
type T is tagged record
    F : Integer := Compute_Default_F;
end record;

function Compute_Default_F return Integer is
begin
    Put_Line ("Compute");
    return 0;
end Compute_Default_F;

V1 : T;
V2 : T := (F => 0);
```

In the declaration of V1, T.F receives a value computed by the subprogram Compute_Default_F. This is part of the default initialization. V2 is initialized manually and thus will not use the default initialization.

For additional expressive power, Ada provides a type called Ada.Finalization. **Controlled** from which you can derive your own type. Then, by overriding the Initialize procedure you can create a constructor for the type:

```
type T is new Ada.Finalization.Controlled with record
    F : Integer;
end record;

procedure Initialize (Self : in out T) is
begin
    Put_Line ("Compute");
    Self.F := 0;
end Initialize;

V1 : T;
V2 : T := (F => 0);
```

Again, this default initialization subprogram is only called for V1; V2 is initialized manually. Furthermore, unlike a C++ or Java constructor, Initialize is a normal subprogram and does not perform any additional initialization such as calling the parent's initialization routines.

When deriving from **Controlled**, it's also possible to override the subprogram Finalize, which is like a destructor and is called for object finalization. Like Initialize, this is a regular subprogram. Do not expect any other finalizers to be automatically invoked for you.

Controlled types also provide functionality that essentially allows overriding the meaning of the assignment operation, and are useful for defining types that manage their own storage reclamation (for example, implementing a reference count reclamation strategy).

55.4 Encapsulation

While done at the class level for C++ and Java, Ada encapsulation occurs at the package level and targets all entities of the language, as opposed to only methods and attributes. For example:

[Ada]

```
package Pck is
    type T is tagged private;
    procedure Method1 (V : T);
private
    type T is tagged record
        F1, F2 : Integer;
    end record;
    procedure Method2 (V : T);
end Pck;
```

[C++]

```
class T {
public:
    virtual void method1 ();
protected:
    int f1, f2;
    virtual void method2 ();
};
```

[Java]

```
public class T {
    public void method1 ();
    protected int f1, f2;
    protected void method2 ();
}
```

The C++ and Java code's use of **protected** and the Ada code's use of **private** here demonstrates how to map these concepts between languages. Indeed, the private part of an Ada child package would have visibility of the private part of its parents, mimicking the notion of **protected**. Only entities declared in the package body are completely isolated from access.

55.5 Abstract Types and Interfaces

Ada, C++ and Java all offer similar functionality in terms of abstract classes, or pure virtual classes. It is necessary in Ada and Java to explicitly specify whether a tagged type or class is **abstract**, whereas in C++ the presence of a pure virtual function implicitly makes the class an abstract base class. For example:

[Ada]

```
package P is
    type T is abstract tagged private;
    procedure Method (Self : T) is abstract;
private
    type T is abstract tagged record
```

(continues on next page)

(continued from previous page)

```

    F1, F2 : Integer;
end record;

end P;
```

[C++]

```

class T {
public:
    virtual void method () = 0;
protected:
    int f1, f2;
};
```

[Java]

```

public abstract class T {
    public abstract void method1 ();
    protected int f1, f2;
};
```

All abstract methods must be implemented when implementing a concrete type based on an abstract type.

Ada doesn't offer multiple inheritance the way C++ does, but it does support a Java-like notion of interfaces. An interface is like a C++ pure virtual class with no attributes and only abstract members. While an Ada tagged type can inherit from at most one tagged type, it may implement multiple interfaces. For example:

[Ada]

```

type Root is tagged record
    F1 : Integer;
end record;
procedure M1 (Self : Root);

type I1 is interface;
procedure M2 (Self : I1) is abstract;

type I2 is interface;
procedure M3 (Self : I2) is abstract;

type Child is new Root and I1 and I2 with record
    F2 : Integer;
end record;

-- M1 implicitly inherited by Child
procedure M2 (Self : Child);
procedure M3 (Self : Child);
```

[C++]

```

class Root {
public:
    virtual void M1();
    int f1;
};

class I1 {
public:
    virtual void M2 () = 0;
```

(continues on next page)

(continued from previous page)

```

};

class I2 {
    public:
        virtual void M3 () = 0;
};

class Child : public Root, I1, I2 {
    public:
        int f2;
        virtual void M2 ();
        virtual void M3 ();
};

```

[Java]

```

public class Root {
    public void M1();
    public int f1;
}

public interface I1 {
    public void M2 () = 0;
}

public class I2 {
    public void M3 () = 0;
}

public class Child extends Root implements I1, I2 {
    public int f2;
    public void M2 ();
    public void M3 ();
}

```

55.6 Invariants

Any private type in Ada may be associated with a Type_Invariant contract. An invariant is a property of a type that must always be true after the return from of any of its primitive subprograms. (The invariant might not be maintained during the execution of the primitive subprograms, but will be true after the return.) Let's take the following example:

```

package Int_List_Pkg is

    type Int_List (Max_Length : Natural) is private
        with Type_Invariant => Is_Sorted (Int_List);

    function Is_Sorted (List : Int_List) return Boolean;

    type Int_Array is array (Positive range <>) of Integer;

    function To_Int_List (Ints : Int_Array) return Int_List;
    function To_Int_Array (List : Int_List) return Int_Array;
    function "&" (Left, Right : Int_List) return Int_List;

```

(continues on next page)

(continued from previous page)

```

... -- Other subprograms
private

type Int_List (Max_Length : Natural) is record
    Length : Natural;
    Data   : Int_Array (1..Max_Length);
end record;

function Is_Sorted (List : Int_List) return Boolean is
    (for all I in List.Data'First .. List.Length-1 =>
     List.Data (I) <= List.Data (I+1));

end Int_List_Pkg;

package body Int_List_Pkg is

procedure Sort (Ints : in out Int_Array) is
begin
    ... Your favorite sorting algorithm
end Sort;

function To_Int_List (Ints : Int_Array) return Int_List is
    List : Int_List :=
        (Max_Length => Ints'Length,
         Length      => Ints'Length,
         Data        => Ints);
begin
    Sort (List.Data);
    return List;
end To_Int_List;

function To_Int_Array (List : Int_List) return Int_Array is
begin
    return List.Data;
end To_Int_Array;

function "&" (Left, Right : Int_List) return Int_List is
    Ints : Int_Array := Left.Data & Right.Data;
begin
    Sort (Ints);
    return To_Int_List (Ints);
end "&";

... -- Other subprograms
end Int_List_Pkg;

```

The Is_Sorted function checks that the type stays consistent. It will be called at the exit of every primitive above. It is permissible if the conditions of the invariant aren't met during execution of the primitive. In To_Int_List for example, if the source array is not in sorted order, the invariant will not be satisfied at the "begin", but it will be checked at the end.

GENERICS

Ada, C++, and Java all have support for generics or templates, but on different sets of language entities. A C++ template can be applied to a class or a function. So can a Java generic. An Ada generic can be either a package or a subprogram.

56.1 Generic Subprograms

In this example, we will swap two generic objects. This is possible in Ada and C++ using a temporary variable. In Java, parameters are a copy of a reference value that is passed into the function, so modifying those references in the function scope has no effect from the caller's context. A generic swap method, like the below Ada or C++ examples is not possible in Java, so we will skip the Java version of this example.

[Ada]

```
generic
    type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
    Temp : A_Type := Left;
begin
    Left := Right;
    Right := Temp;
end Swap;
```

[C++]

```
template <class AType>
AType swap (AType & left, AType & right) {
    AType temp = left;
    left = right;
    right = temp;
}
```

And examples of using these:

[Ada]

```
declare
    type R is record
        F1, F2 : Integer;
    end record;

    procedure Swap_R is new Swap (R);
        A, B : R;
begin
    ...
```

(continues on next page)

(continued from previous page)

```
Swap_R (A, B);
end;
```

[C++]

```
class R {
    public:
        int f1, f2;
};

R a, b;
...
swap (a, b);
```

The C++ template becomes usable once defined. The Ada generic needs to be explicitly instantiated using a local name and the generic's parameters.

56.2 Generic Packages

Next, we're going to create a generic unit containing data and subprograms. In Java or C++, this is done through a class, while in Ada, it's a *generic package*. The Ada and C++ model is fundamentally different from the Java model. Indeed, upon instantiation, Ada and C++ generic data are duplicated; that is, if they contain global variables (Ada) or static attributes (C++), each instance will have its own copy of the variable, properly typed and independent from the others. In Java, generics are only a mechanism to have the compiler do consistency checks, but all instances are actually sharing the same data where the generic parameters are replaced by *java.lang.Object*. Let's look at the following example:

[Ada]

```
generic
    type T is private;
package Gen is
    type C is tagged record
        V : T;
    end record;

    G : Integer;
end Gen;
```

[C++]

```
template <class T>
class C{
    public:
        T v;
        static int G;
};
```

[Java]

```
public class C <T> {
    public T v;
    public static int G;
}
```

In all three cases, there's an instance variable (v) and a static variable (G). Let's now look at the behavior (and syntax) of these three instantiations:

[Ada]

```
declare
    package I1 is new Gen (Integer);
    package I2 is new Gen (Integer);
    subtype Str10 is String (1..10);
    package I3 is new Gen (Str10);
begin
    I1.G := 0;
    I2.G := 1;
    I3.G := 2;
end;
```

[C++]

```
C <int>::G = 0;
C <int>::G = 1;
C <char *>::G = 2;
```

[Java]

```
C.G = 0;
C.G = 1;
C.G = 2;
```

In the Java case, we access the generic entity directly without using a parametric type. This is because there's really only one instance of C, with each instance sharing the same global variable G. In C++, the instances are implicit, so it's not possible to create two different instances with the same parameters. The first two assignments are manipulating the same global while the third one is manipulating a different instance. In the Ada case, the three instances are explicitly created, named, and referenced individually.

56.3 Generic Parameters

Ada offers a wide variety of generic parameters which is difficult to translate into other languages. The parameters used during instantiation — and as a consequence those on which the generic unit may rely on — may be variables, types, or subprograms with certain properties. For example, the following provides a sort algorithm for any kind of array:

```
generic
    type Component is private;
    type Index is (<>);
    with function "<" (Left, Right : Component) return Boolean;
    type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

The above declaration states that we need a type (Component), a discrete type (Index), a comparison subprogram ("<"), and an array definition (Array_Type). Given these, it's possible to write an algorithm that can sort any Array_Type. Note the usage of the **with** reserved word in front of the function name, to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```
type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type, such as String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<>); -- T is a discrete type
```

(continues on next page)

(continued from previous page)

```
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type, T2 is its designated type
```

EXCEPTIONS

Exceptions are a mechanism for dealing with run-time occurrences that are rare, that usually correspond to errors (such as improperly formed input data), and whose occurrence causes an unconditional transfer of control.

57.1 Standard Exceptions

Compared with Java and C++, the notion of an Ada exception is very simple. An exception in Ada is an object whose "type" is **exception**, as opposed to classes in Java or any type in C++. The only piece of user data that can be associated with an Ada exception is a String. Basically, an exception in Ada can be raised, and it can be handled; information associated with an occurrence of an exception can be interrogated by a handler.

Ada makes heavy use of exceptions especially for data consistency check failures at run time. These include, but are not limited to, checking against type ranges and array boundaries, null pointers, various kind of concurrency properties, and functions not returning a value. For example, the following piece of code will raise the exception `Constraint_Error`:

```
procedure P is
    V : Positive;
begin
    V := -1;
end P;
```

In the above code, we're trying to assign a negative value to a variable that's declared to be positive. The range check takes place during the assignment operation, and the failure raises the `Constraint_Error` exception at that point. (Note that the compiler may give a warning that the value is out of range, but the error is manifest as a run-time exception.) Since there is no local handler, the exception is propagated to the caller; if `P` is the main procedure, then the program will be terminated.

Java and C++ can **throw** and **catch** exceptions when **trying** code. All Ada code is already implicitly within **try** blocks, and exceptions are raised and handled.

[Ada]

```
begin
    Some_Call;
exception
    when Exception_1 =>
        Put_Line ("Error 1");
    when Exception_2 =>
        Put_Line ("Error 2");
    when others =>
        Put_Line ("Unknown error");
end;
```

[C++]

```
try {
    someCall ();
} catch (Exception1) {
    cout << "Error 1" << endl;
} catch (Exception2) {
    cout << "Error 2" << endl;
} catch (...) {
    cout << "Unknown error" << endl;
}
```

[Java]

```
try {
    someCall ();
} catch (Exception1 e1) {
    System.out.println ("Error 1");
} catch (Exception2 e2) {
    System.out.println ("Error 2");
} catch (Throwable e3) {
    System.out.println ("Unknown error");
}
```

Raising and throwing exceptions is permissible in all three languages.

57.2 Custom Exceptions

Custom exception declarations resemble object declarations, and they can be created in Ada using the **exception** keyword:

```
My_Exception : exception;
```

Your exceptions can then be raised using a **raise** statement, optionally accompanied by a message following the **with** reserved word:

[Ada]

```
raise My_Exception with "Some message";
```

[C++]

```
throw My_Exception ("Some message");
```

[Java]

```
throw new My_Exception ("Some message");
```

Language defined exceptions can also be raised in the same manner:

```
raise Constraint_Error;
```

CONCURRENCY

58.1 Tasks

Java and Ada both provide support for concurrency in the language. The C++ language has added a concurrency facility in its most recent revision, C++11, but we are assuming that most C++ programmers are not (yet) familiar with these new features. We thus provide the following mock API for C++ which is similar to the Java Thread class:

```
class Thread {
public:
    virtual void run (); // code to execute
    void start (); // starts a thread and then call run ()
    void join (); // waits until the thread is finished
};
```

Each of the following examples will display the 26 letters of the alphabet twice, using two concurrent threads/tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

```
procedure Main is -- implicitly called by the environment task
task My_Task;

task body My_Task is
begin
    for I in 'A' .. 'Z' loop
        Put_Line (I);
    end loop;
end My_Task;
begin
    for I in 'A' .. 'Z' loop
        Put_Line (I);
    end loop;
end Main;
```

[C++]

```
class MyThread : public Thread {
public:

    void run () {
        for (char i = 'A'; i <= 'Z'; ++i) {
            cout << i << endl;
        }
    }
};
```

(continues on next page)

(continued from previous page)

```
int main (int argc, char ** argv) {
    MyThread myTask;
    myTask.start ();

    for (char i = 'A'; i <= 'Z'; ++i) {
        cout << i << endl;
    }

    myTask.join ();

    return 0;
}
```

[Java]

```
public class Main {
    static class MyThread extends Thread {
        public void run () {
            for (char i = 'A'; i <= 'Z'; ++i) {
                System.out.println (i);
            }
        }
    }

    public static void main (String args) {
        MyThread myTask = new MyThread ();
        myTask.start ();

        for (char i = 'A'; i <= 'Z'; ++i) {
            System.out.println (i);
        }
        myTask.join ();
    }
}
```

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the **begin**. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

An Ada task type is somewhat analogous to a Java Thread subclass, but in Java the instances of such a subclass are always dynamically allocated. In Ada an instance of a task type may either be declared or dynamically allocated.

Task types can be parametrized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the implementation's task scheduling algorithm.

[Ada]

```
task type My_Task (First : Character);

task body My_Task is
begin
    for I in First .. 'Z' loop
```

(continues on next page)

(continued from previous page)

```

    Put_Line (I);
end loop;
end My_Task;

procedure Main is
  Tab : array (0 .. 9) of My_Task ('G');
begin
  null;
end Main;

```

[C++]

```

class MyThread : public Thread {
public:

  char first;

  void run () {
    for (char i = first; i <= 'Z'; ++i) {
      cout << i << endl;
    }
  }
};

int main (int argc, char ** argv) {
  MyThread tab [10];

  for (int i = 0; i < 9; ++i) {
    tab [i].first = 'G';
    tab [i].start ();
  }

  for (int i = 0; i < 9; ++i) {
    tab [i].join ();
  }

  return 0;
}

```

[Java]

```

public class MyThread extends Thread {
  public char first;

  public MyThread (char first){
    this.first = first;
  }

  public void run () {
    for (char i = first; i <= 'Z'; ++i) {
      cout << i << endl;
    }
  }
}

public class Main {
  public static void main (String args) {
    MyThread [] tab = new MyThread [10];

    for (int i = 0; i < 9; ++i) {
      tab [i] = new MyThread ('G');
    }
  }
}

```

(continues on next page)

(continued from previous page)

```

        tab [i].start ();
    }

    for (int i = 0; i < 9; ++i) {
        tab [i].join ();
    }
}
}

```

In Ada a task may be allocated on the heap as opposed to the stack. The task will then start as soon as it has been allocated, and terminates when its work is completed. This model is probably the one that's the most similar to Java:

[Ada]

```

type Ptr_Task is access My_Task;

procedure Main is
    T : Ptr_Task;
begin
    T := new My_Task ('G');
end Main;

```

[C++]

```

int main (int argc, char ** argv) {
    MyThread * t = new MyThread ();
    t->first = 'G';
    t->start ();
    return 0;
}

```

[Java]

```

public class Main {
    public static void main (String args) {
        MyThread t = new MyThread ('G');

        t.start ();
    }
}

```

58.2 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Ada's rendezvous facility cannot be modeled with C++ or Java without complex machinery. Therefore, this section will just show examples written in Ada.

Let's consider the following example:

```

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    task After is
        entry Go;
    end After ;

```

(continues on next page)

(continued from previous page)

```

task body After is
begin
    accept Go;
    Put_Line ("After");
end After;

begin
    Put_Line ("Before");
    After.Go;
end;

```

The Go **entry** declared in After is the external interface to the task. In the task body, the **accept** statement causes the task to wait for a call on the entry. This particular **entry** and **accept** pair doesn't do much more than cause the task to wait until Main calls After.Go. So, even though the two tasks start simultaneously and execute independently, they can coordinate via Go. Then, they both continue execution independently after the rendezvous.

The **entry/accept** pair can take/pass parameters, and the **accept** statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

```

with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

    task After is
        entry Go (Text : String);
    end After ;

    task body After is
begin
    accept Go (Text : String) do
        Put_Line ("After: " & Text);
    end Go;
end After;

begin
    Put_Line ("Before");
    After.Go ("Main");
end;

```

In the above example, the Put_Line is placed in the **accept** statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the **begin** of Main, task After is started and the main procedure is suspended.
2. After reaches the **accept** statement and is suspended, since there is no pending call on the Go entry.
3. The main procedure is awakened and executes the Put_Line invocation, displaying the string "Before".
4. The main procedure calls the Go entry. Since After is suspended on its **accept** statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task After is awakened to execute the body of the **accept** statement. The actual parameter "**Main**" is passed to the **accept** statement, and the Put_Line invocation is executed. As a result, the string "**After: Main**" is displayed.
6. When the **accept** statement is completed, both the After task and the main proce-

dure are ready to run. Suppose that the Main procedure is given the processor. It reaches its **end**, but the local task After has not yet terminated. The main procedure is suspended.

7. The After task continues, and terminates since it is at its **end**. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.

58.3 Selective Rendezvous

The accept statement by itself can only wait for a single event (call) at a time. The **select** statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call Increment, Decrement, and Get:

```
task Counter is
  entry Get (Result : out Integer);
  entry Increment;
  entry Decrement;
end Counter;

task body Counter is
  Value : Integer := 0;
begin
  loop
    select
      accept Increment do
        Value := Value + 1;
      end Increment;
    or
      accept Decrement do
        Value := Value - 1;
      end Decrement;
    or
      accept Get (Result : out Integer) do
        Result := Value;
      end Get;
    or
      delay 60.0; -- delay 1 minute
      exit;
    end select;
  end loop;
end Counter;
```

When the task's statement flow reaches the **select**, it will wait for all four events — three entries and a delay — in parallel. If the delay of one minute is exceeded, the task will execute the statements following the **delay** statement (and in this case will exit the loop, in effect terminating the task). The accept bodies for the Increment, Decrement, or Get entries will be otherwise executed as they're called. These four sections of the **select** statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a "race condition" could occur where multiple tasks simultaneously calling, say, Increment, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call Increment then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and Counter is not ready to accept the call (i.e., if Counter is not suspended

at the `select` statement) then the calling task is suspended, and placed in the queue of the entry that it is calling. From the perspective of the Counter task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case Counter is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the one minute delay (whose effect was noted above).
- There is a call pending on exactly one of the entries. In this case control passes to the `select` branch with an `accept` statement for that entry. The choice of which caller to accept, if more than one, depends on the queuing policy, which can be specified via a pragma defined in the Real-Time Systems Annex of the Ada standard; the default is First-In First-Out.
- There are calls pending on more than one entry. In this case one of the entries with pending callers is chosen, and then one of the callers is chosen to be de-queued (the choices depend on the queueing policy).

58.4 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared data object, an alternative (and generally preferable) style is through a *protected object*, an efficiently implementable mechanism that makes the effect more explicit. A protected object has a public interface (its *protected operations*) for accessing and manipulating the object's components (its private part). Mutual exclusion is enforced through a conceptual lock on the object, and encapsulation ensures that the only external access to the components are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by procedures or entries, and read-only operations by functions. The lock mechanism is implemented so that it's possible to perform concurrent read operations but not concurrent write or read/write operations.

Let's reimplement our earlier tasking example with a protected object called Counter:

```
protected Counter is
    function Get return Integer;
    procedure Increment;
    procedure Decrement;
private
    Value : Integer := 0;
end Counter;

protected body Counter is
    function Get return Integer is
    begin
        return Value;
    end Get;

    procedure Increment is
    begin
        Value := Value + 1;
    end Increment;

    procedure Decrement is
    begin
        Value := Value - 1;
    end Decrement;
end Counter;
```

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

```
Counter.Increment;  
Counter.Decrement;  
Put_Line (Integer'Image (Counter.Get));
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```
declare  
  protected type Counter is  
    -- as above  
  end Counter;  
  
  protected body Counter is  
    -- as above  
  end Counter;  
  
  C1 : Counter;  
  C2 : Counter;  
begin  
  C1.Increment;  
  C2.Decrement;  
  ...  
end;
```

Protected objects and types can declare a procedure-like operation known as an "entry". An entry is somewhat similar to a procedure but includes a so-called *barrier condition* that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

```
protected type Binary_Semaphore is  
  entry Wait;  
  procedure Signal;  
private  
  Signaled : Boolean := False;  
end Binary_Semaphore;  
  
protected body Binary_Semaphore is  
  entry Wait when Signaled is  
  begin  
    Signaled := False;  
  end Wait;  
  
  procedure Signal is
```

(continues on next page)

(continued from previous page)

```
begin
  Signaled := True;
end Signal;
end Binary_Semaphore;
```

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the *References* section.

LOW LEVEL PROGRAMMING

59.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```
type R is record
    V : Integer range 0 .. 255;
    B1 : Boolean;
    B2 : Boolean;
end record
with Pack;
```

[C++]

```
struct R {
    unsigned int v:8;
    bool b1;
    bool b2;
};
```

[Java]

```
public class R {
    public byte v;
    public boolean b1;
    public boolean b2;
}
```

The Ada and the C++ code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C++ and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C++, the number of bits required by each field needs to be specified. Here, we're stating that v is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The Pack

aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

```
type R is record
  V : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record;

for R use record
  -- Occupy the first bit of the first byte.
  B1 at 0 range 0 .. 0;

  -- Occupy the last 7 bits of the first byte,
  -- as well as the first bit of the second byte.
  V at 0 range 1 .. 8;

  -- Occupy the second bit of the second byte.
  B2 at 1 range 1 .. 1;
end record;
```

We omit the **with** Pack directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type R across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

59.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the rdtsc instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64 bit value split between the edx and eax registers.

GNAT provides a subprogram called System.Machine_Code.Asm that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

```
with System.Machine_Code; use System.Machine_Code;
with Interfaces;           use Interfaces;

function Get_Processor_Cycles return Unsigned_64 is
  Low, High : Unsigned_32;
  Counter   : Unsigned_64;
begin
  Asm ("rdtsc",
        Outputs =>
        (Unsigned_32'Asm_Output ("=a", Low),
```

(continues on next page)

(continued from previous page)

```

    Unsigned_32'Asm_Output ("=d", High)),
Volatile => True);

Counter :=
  Unsigned_64 (High) * 2 ** 32 +
  Unsigned_64 (Low);

return Counter;
end Get_Processor_Cycles;

```

The Unsigned_32'Asm_Output clauses above provide associations between machine registers and source-level variables to be updated. "=a" and "=d" refer to the eax and edx machine registers, respectively. The use of the Unsigned_32 and Unsigned_64 types from package Interfaces ensures correct representation of the data. We assemble the two 32-bit values to form a single 64 bit value.

We set the Volatile parameter to **True** to tell the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the eax and edx registers to implement the High and Low variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

59.3 Interfacing with C

Much effort was spent making Ada easy to interface with other languages. The Interfaces package hierarchy and the pragmas Convention, Import, and Export allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

```

struct my_struct {
    int A, B;
};

void call (my_struct * p) {
    printf ("%d", p->A);
}

```

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C **struct my_struct** is represented, we can use the following record along with a **pragma Convention**. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

```

type my_struct is record
    A : Interfaces.C.int;
    B : Interfaces.C.int;
end record;
pragma Convention (C, my_struct);

```

Describing a foreign subprogram call to Ada code is called "binding" and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a **void** function maps to an

Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a `pragma Import`:

```
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The `Import` pragma specifies that whenever `Call` is invoked by Ada code, it should invoke the `call` function with the C calling convention.

And that's all that's necessary. Here's an example of a call to `Call`:

```
declare
  V : my_struct := (A => 1, B => 2);
begin
  Call (V);
end;
```

You can also make Ada subprograms available to C code, and examples of this can be found in the GNAT User's Guide. Interfacing with C++ and Java use implementation-dependent features that are also available with GNAT.

CONCLUSION

All the usual paradigms of imperative programming can be found in all three languages that we surveyed in this document. However, Ada is different from the rest in that it's more explicit when expressing properties and expectations. This is a good thing: being more formal affords better communication among programmers on a team and between programmers and machines. You also get more assurance of the coherence of a program at many levels. Ada can help reduce the cost of software maintenance by shifting the effort to creating a sound system the first time, rather than working harder, more often, and at greater expense, to fix bugs found later in systems already in production. Applications that have reliability needs, long term maintenance requirements, or safety/security concerns are those for which Ada has a proven track record.

It's becoming increasingly common to find systems implemented in multiple languages, and Ada has standard interfacing facilities to allow Ada code to invoke subprograms and/or reference data structures from other language environments, or vice versa. Use of Ada thus allows easy interfacing between different technologies, using each for what it's best at.

We hope this guide has provided some insight into the Ada software engineer's world and has made Ada more accessible to programmers already familiar with programming in other languages.

CHAPTER
SIXTYONE

REFERENCES

The Ada Information Clearinghouse website <http://www.adaic.org/learn/materials/>, maintained by the Ada Resource Association, contains links to a variety of training materials (books, articles, etc.) that can help in learning Ada. The Development Center page <http://www.adacore.com/knowledge> on AdaCore's website also contains links to useful information including vides and tutorials on Ada.

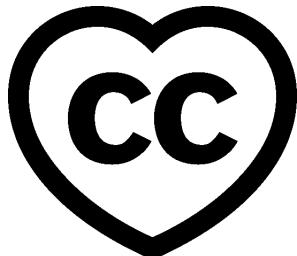
The most comprehensive textbook is John Barnes' *Programming in Ada 2012*, which is oriented towards professional software developers.

Part VI

Ada for the Embedded C Developer

Copyright © 2020 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](#)⁹³



This course introduces you to the Ada language by comparing it to C. It assumes that you have good knowledge of the C language. It also assumes that the choice of learning Ada is guided by considerations linked to reliability, safety or security. In that sense, it teaches you Ada paradigms that should be applied in replacement of those usually applied in C.

This course also introduces you to the SPARK subset of the Ada programming language, which removes a few features of the language with undefined behavior, so that the code is fit for sound static analysis techniques.

This course was written by Quentin Ochem, Robert Tice, Gustavo A. Hoffmann, and Patrick Rogers and reviewed by Patrick Rogers, Filip Gajowniczek, and Tucker Taft.

⁹³ <http://creativecommons.org/licenses/by-sa/4.0>

INTRODUCTION

62.1 So, what is this Ada thing anyway?

To answer this question let's introduce Ada as it compares to C for an embedded application. C developers are used to a certain coding semantic and style of programming. Especially in the embedded domain, developers are used to working at a very low level near the hardware to directly manipulate memory and registers. Normal operations involve mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations as it is a low level language that was designed to replace assembly language for faster, more efficient programming. Because of this minimal abstraction, the programmer has to model the data that represents the problem they are trying to solve using the language of the physical hardware.

Let's look at an example of this problem in action by comparing the same program in Ada and C:

[C]

Listing 1: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define DEGREES_MAX          (360)
5 typedef unsigned int degrees;
6
7 #define MOD_DEGREES(x)    (x % DEGREES_MAX)
8
9 degrees add_angles(degrees* list, int length)
10 {
11     degrees sum = 0;
12     for(int i = 0; i < length; ++i) {
13         sum += list[i];
14     }
15
16     return sum;
17 }
18
19 int main(int argc, char** argv)
20 {
21     degrees list[argc - 1];
22
23     for(int i = 1; i < argc; ++i) {
24         list[i - 1] = MOD_DEGREES(atoi(argv[i]));
25     }
26
27     printf("Sum: %d\n", add_angles(list, argc - 1));
28

```

(continues on next page)

(continued from previous page)

```
29     return 0;
30 }
```

Runtime output

```
Sum: 0
```

```
[Ada]
```

Listing 2: sum_angles.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Sum_Angles is
5
6      DEGREES_MAX : constant := 360;
7      type Degrees is mod DEGREES_MAX;
8
9      type Degrees_List is array (Natural range <>) of Degrees;
10
11     function Add_Angles (List : Degrees_List) return Degrees
12     is
13         Sum : Degrees := 0;
14     begin
15         for I in List'Range loop
16             Sum := Sum + List (I);
17         end loop;
18
19         return Sum;
20     end Add_Angles;
21
22     List : Degrees_List (1 .. Argument_Count);
23 begin
24     for I in List'Range loop
25         List (I) := Degrees (Integer'Value (Argument (I)));
26     end loop;
27
28     Put_Line ("Sum:" & Add_Angles (List)'Img);
29 end Sum_Angles;
```

Runtime output

```
Sum: 0
```

Here we have a piece of code in C and in Ada that takes some numbers from the command line and stores them in an array. We then sum all of the values in the array and print the result. The tricky part here is that we are working with values that model an angle in degrees. We know that angles are modular types, meaning that angles greater than 360° can also be represented as Angle $\text{mod } 360$. So if we have an angle of 400° , this is equivalent to 40° . In order to model this behavior in C we had to create the MOD_DEGREES macro, which performs the modulus operation. As we read values from the command line, we convert them to integers and perform the modulus before storing them into the array. We then call add_angles which returns the sum of the values in the array. Can you spot the problem with the C code?

Try running the Ada and C examples using the input sequence `340 2 50 70`. What does the C program output? What does the Ada program output? Why are they different?

The problem with the C code is that we forgot to call MOD_DEGREES in the for loop of add_angles. This means that it is possible for add_angles to return values greater than

DEGREES_MAX. Let's look at the equivalent Ada code now to see how Ada handles the situation. The first thing we do in the Ada code is to create the type Degrees which is a modular type. This means that the compiler is going to handle performing the modulus operation for us. If we use the same for loop in the Add_Angles function, we can see that we aren't doing anything special to make sure that our resulting value is within the 360° range we need it to be in.

The takeaway from this example is that Ada tries to abstract some concepts from the developer so that the developer can focus on solving the problem at hand using a data model that models the real world rather than using data types prescribed by the hardware. The main benefit of this is that the compiler takes some responsibility from the developer for generating correct code. In this example we forgot to put in a check in the C code. The compiler inserted the check for us in the Ada code because we told the compiler what we were trying to accomplish by defining strong types.

Ideally, we want all the power that the C programming language can give us to manipulate the hardware we are working on while also allowing us the ability to more accurately model data in a safe way. So, we have a dilemma; what can give us the power of operations like the C language, but also provide us with features that can minimize the potential for developer error? Since this course is about Ada, it's a good bet we're about to introduce the Ada language as the answer to this question...

Unlike C, the Ada language was designed as a higher level language from its conception; giving more responsibility to the compiler to generate correct code. As mentioned above, with C, developers are constantly shifting, masking, and accessing bits directly on memory pointers. In Ada, all of these operations are possible, but in most cases, there is a better way to perform these operations using higher level constructs that are less prone to mistakes, like off-by-one or unintentional buffer overflows. If we were to compare the same application written using C and with Ada using high level constructs, we would see similar performance in terms of speed and memory efficiency. If we compare the object code generated by both compilers, it's possible that they even look identical!

62.2 Ada — The Technical Details

Like C, Ada is a compiled language. This means that the compiler will parse the source code and emit machine code native to the target hardware. The Ada compiler we will be discussing in this course is the GNAT compiler. This compiler is based on the GCC technology like many C and C++ compilers available. When the GNAT compiler is invoked on Ada code, the GNAT front-end expands and translates the Ada code into an intermediate language which is passed to GCC where the code is optimized and translated to machine code. A C compiler based on GCC performs the same steps and uses the same intermediate GCC representation. This means that the optimizations we are used to seeing with a GCC based C compiler can also be applied to Ada code. The main difference between the two compilers is that the Ada compiler is expanding high level constructs into intermediate code. After expansion, the Ada code will be very similar to the equivalent C code.

It is possible to do a line-by-line translation of C code to Ada. This feels like a natural step for a developer used to C paradigms. However, there may be very little benefit to doing so. For the purpose of this course, we're going to assume that the choice of Ada over C is guided by considerations linked to reliability, safety or security. In order to improve upon the reliability, safety and security of our application, Ada paradigms should be applied in replacement of those usually applied in C. Constructs such as pointers, preprocessor macros, bitwise operations and defensive code typically get expressed in Ada in very different ways, improving the overall reliability and readability of the applications. Learning these new ways of coding, often, requires effort by the developer at first, but proves more efficient once the paradigms are understood.

In this course we will also introduce the SPARK subset of the Ada programming language.

The SPARK subset removes a few features of the language, i.e., those that make proof difficult, such as pointer aliasing. By removing these features we can write code that is fit for sound static analysis techniques. This means that we can run mathematical provers on the SPARK code to prove certain safety or security properties about the code.

THE C DEVELOPER'S PERSPECTIVE ON ADA

63.1 What we mean by Embedded Software

The Ada programming language is a general programming language, which means it can be used for many different types of applications. One type of application where it particularly shines is reliable and safety-critical embedded software; meaning, a platform with a microprocessor such as ARM, PowerPC, x86, or RISC-V. The application may be running on top of an embedded operating system, such as an embedded Linux, or directly on bare metal. And the application domain can range from small entities such as firmware or device controllers to flight management systems, communication based train control systems, or advanced driver assistance systems.

63.2 The GNAT Toolchain

The toolchain used throughout this course is called GNAT, which is a suite of tools with a compiler based on the GCC environment. It can be obtained from AdaCore, either as part of a commercial contract with [GNAT Pro⁹⁴](#) or at no charge with the [GNAT Community edition⁹⁵](#). The information in this course will be relevant no matter which edition you're using. Most examples will be runnable on the native Linux or Windows version for convenience. Some will only be relevant in the context of a cross toolchain, in which case we'll be using the embedded ARM bare metal toolchain.

As for any Ada compiler, GNAT takes advantage of implementation permissions and offers a project management system. Because we're talking about embedded platforms, there are a lot of topics that we'll go over which will be specific to GNAT, and sometimes to specific platforms supported by GNAT. We'll try to make the distinction between what is GNAT-specific and Ada generic as much as possible throughout this course.

For an introduction to the GNAT Toolchain for the GNAT Community edition, you may refer to the [*Introduction to GNAT Toolchain*](#) (page 935) course.

⁹⁴ <https://www.adacore.com/gnatpro>

⁹⁵ <https://www.adacore.com/community>

63.3 The GNAT Toolchain for Embedded Targets

When we're discussing embedded programming, our target device is often different from the host, which is the device we're using to actually write and build an application. In this case, we're talking about cross compilation platforms (concisely referred to as cross platforms).

The GNAT toolchain supports cross platform compilation for various target devices. This section provides a short introduction to the topic. For more details, please refer to the [GNAT User's Guide Supplement for Cross Platforms⁹⁶](#)

GNAT supports two types of cross platforms:

- **cross targets**, where the target device has an embedded operating system.
 - ARM-Linux, which is commonly found in a Raspberry-Pi, is a prominent example.
- **bareboard targets**, where the run-times do not depend on an operating system.
 - In this case, the application has direct access to the system hardware.

For each platform, a set of run-time libraries is available. Run-time libraries implement a subset of the Ada language for different use cases, and they're different for each target platform. They may be selected via an attribute in the project's GPR project file or as a command-line switch to **GPRbuild**. Although the run-time libraries may vary from target to target, the user interface stays the same, providing portability for the application.

Run-time libraries consists of:

1. Files that are dependent on the target board.
 - These files are responsible for configuring and interacting with the hardware.
 - They are known as a Board Support Package — commonly referred to by their abbreviation *BSP*.
2. Code that is target-independent.
 - This code implements language-defined functionality.

The bareboard run-time libraries are provided as customized run-times that are configured to target a very specific micro-controller or processor. Therefore, for different micro-controllers and processors, the run-time libraries need to be ported to the specific target. These are some examples of what needs to be ported:

- startup code / scripts;
- clock frequency initializations;
- memory mapping / allocation;
- interrupts and interrupt priorities;
- register descriptions.

For more details on the topic, please refer to the following chapters of the [GNAT User's Guide Supplement for Cross Platforms⁹⁷](#):

- [Bareboard Topics⁹⁸](#)
- [Customized Run-Time Libraries⁹⁹](#)

⁹⁶ https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html

⁹⁷ https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx.html

⁹⁸ http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx/bareboard_topics.html

⁹⁹ http://docs.adacore.com/live/wave/gnat_ugx/html/gnat_ugx/gnat_ugx/customized_run-time_libraries.html

63.4 Hello World in Ada

The first piece of code to translate from C to Ada is the usual Hello World program:

[C]

Listing 1: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     printf("Hello World\n");
6     return 0;
7 }
```

Runtime output

Hello World

[Ada]

Listing 2: hello_world.adb

```

1 with Ada.Text_IO;
2
3 procedure Hello_World
4 is
5 begin
6     Ada.Text_IO.Put_Line ("Hello World");
7 end Hello_World;
```

Runtime output

Hello World

The resulting program will print Hello World on the screen. Let's now dissect the Ada version to describe what is going on:

The first line of the Ada code is giving us access to the Ada.Text_IO library which contains the Put_Line function we will use to print the text to the console. This is similar to C's `#include <stdio.h>`. We then create a procedure which executes Put_Line which prints to the console. This is similar to C's printf function. For now, we can assume these Ada and C features have similar functionality. In reality, they are very different. We will explore that more as we delve further into the Ada language.

You may have noticed that the Ada syntax is more verbose than C. Instead of using braces {} to declare scope, Ada uses keywords. `is` opens a declarative scope — which is empty here as there's no variable to declare. `begin` opens a sequence of statements. Within this sequence, we're calling the function Put_Line, prefixing explicitly with the name of the library unit where it's declared, Ada.Text_IO. The absence of the end of line \n can also be noted, as Put_Line always terminates by an end of line.

63.5 The Ada Syntax

Ada syntax might seem peculiar at first glance. Unlike many other languages, it's not derived from the popular C style of notation with its ample use of brackets; rather, it uses a more expository syntax coming from Pascal. In many ways, Ada is a more explicit language — its syntax was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner. For example:

- full words like **begin** and **end** are used in place of curly braces.
- Conditions are written using **if**, **then**, **elsif**, **else**, and **end if**.
- Ada's assignment operator does not double as an expression, eliminating potential mistakes that could be caused by = being used where == should be.

All languages provide one or more ways to express comments. In Ada, two consecutive hyphens -- mark the start of a comment that continues to the end of the line. This is exactly the same as using // for comments in C. Multi line comments like C's /* */ do not exist in Ada.

Ada compilers are stricter with type and range checking than most C programmers are used to. Most beginning Ada programmers encounter a variety of warnings and error messages when coding, but this helps detect problems and vulnerabilities at compile time — early on in the development cycle. In addition, checks (such as array bounds checks) provide verification that could not be done at compile time but can be performed either at run-time, or through formal proof (with the SPARK tooling).

Ada identifiers and reserved words are case insensitive. The identifiers VAR, var and VaR are treated as the same identifier; likewise **begin**, **BEGIN**, **Begin**, etc. Identifiers may include letters, digits, and underscores, but must always start with a letter. There are 73 reserved keywords in Ada that may not be used as identifiers, and these are:

abort	else	null	select
abs	elsif	of	separate
abstract	end	or	some
accept	entry	others	subtype
access	exception	out	synchronized
aliased	exit	overriding	tagged
all	for	package	task
and	function	pragma	terminate
array	generic	private	then
at	goto	procedure	type
begin	if	protected	until
body	in	raise	use
case	interface	range	when
constant	is	record	while
declare	limited	rem	with
delay	loop	renames	xor
delta	mod	requeue	
digits	new	return	
do	not	reverse	

63.6 Compilation Unit Structure

Both C and Ada were designed with the idea that the code specification and code implementation could be separated into two files. In C, the specification typically lives in the .h, or header file, and the implementation lives in the .c file. Ada is superficially similar to C. With the GNAT toolchain, compilation units are stored in files with an .ads extension for specifications and with an .adb extension for implementations.

One main difference between the C and Ada compilation structure is that Ada compilation units are structured into something called packages.

63.7 Packages

The package is the basic modularization unit of the Ada language, as is the class for Java and the header and implementation pair for C. A specification defines a package and the implementation implements the package. We saw this in an earlier example when we included the Ada.Text_Io package into our application. The package specification has the structure:

[Ada]

```
-- my_package.ads
package My_Package is
    -- public declarations

    private
        -- private declarations
end My_Package;
```

The package implementation, or body, has the structure:

```
-- my_package.adb
package body My_Package is
    -- implementation
end My_Package;
```

63.7.1 Declaration Protection

An Ada package contains three parts that, for GNAT, are separated into two files: .ads files contain public and private Ada specifications, and .adb files contain the implementation, or Ada bodies.

[Ada]

```
package Package_Name is
    -- public specifications
private
    -- private specifications
end Package_Name;

package body Package_Name is
```

(continues on next page)

(continued from previous page)

```
-- implementation  
end Package_Name;
```

Private types are useful for preventing the users of a package's types from depending on the types' implementation details. Another use-case is the prevention of package users from accessing package state/data arbitrarily. The private reserved word splits the package spec into *public* and *private* parts. For example:

[Ada]

Listing 3: types.ads

```
1 package Types is  
2     type Type_1 is private;  
3     type Type_2 is private;  
4     type Type_3 is private;  
5     procedure P (X : Type_1);  
6     -- ...  
7     private  
8         procedure Q (Y : Type_1);  
9         type Type_1 is new Integer range 1 .. 1000;  
10        type Type_2 is array (Integer range 1 .. 1000) of Integer;  
11        type Type_3 is record  
12            A, B : Integer;  
13        end record;  
14    end Types;
```

Subprograms declared above the **private** separator (such as P) will be visible to the package user, and the ones below (such as Q) will not. The body of the package, the implementation, has access to both parts. A package specification does not require a private section.

63.7.2 Hierarchical Packages

Ada packages can be organized into hierarchies. A child unit can be declared in the following way:

[Ada]

```
-- root-child.ads  
  
package Root.Child is  
    -- package spec goes here  
end Root.Child;  
  
-- root-child.adb  
  
package body Root.Child is  
    -- package body goes here  
end Root.Child;
```

Here, Root.Child is a child package of Root. The public part of Root.Child has access to the public part of Root. The private part of Child has access to the private part of Root, which is one of the main advantages of child packages. However, there is no visibility relationship between the two bodies. One common way to use this capability is to define subsystems around a hierarchical naming scheme.

63.7.3 Using Entities from Packages

Entities declared in the visible part of a package specification can be made accessible using a **with** clause that references the package, which is similar to the C **#include** directive. After a **with** clause makes a package available, references to the package contents require the name of the package as a prefix, with a dot after the package name. This prefix can be omitted if a **use** clause is employed.

[Ada]

Listing 4: pck.ads

```

1  --  pck.ads
2
3  package Pck is
4      My_Glob : Integer;
5  end Pck;
```

Listing 5: main.adb

```

1  --  main.adb
2
3  with Pck;
4
5  procedure Main is
6  begin
7      Pck.My_Glob := 0;
8  end Main;
```

In contrast to C, the Ada **with** clause is a *semantic inclusion* mechanism rather than a *text inclusion* mechanism; for more information on this difference please refer to [Packages](#) (page 31).

63.8 Statements and Declarations

The following code samples are all equivalent, and illustrate the use of comments and working with integer variables:

[C]

Listing 6: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     // variable declarations
6     int a = 0, b = 0, c = 100, d;
7
8     // c shorthand for increment
9     a++;
10
11    // regular addition
12    d = a + b + c;
13
14    // printing the result
15    printf("d = %d\n", d);
16
```

(continues on next page)

(continued from previous page)

```
17    return 0;  
18 }
```

Runtime output

```
d = 101
```

[Ada]

Listing 7: main.adb

```
1 with Ada.Text_IO;  
2  
3 procedure Main  
4 is  
5     -- variable declaration  
6     A, B : Integer := 0;  
7     C      : Integer := 100;  
8     D      : Integer;  
9 begin  
10    -- Ada does not have a shortcut format for increment like in C  
11    A := A + 1;  
12  
13    -- regular addition  
14    D := A + B + C;  
15  
16    -- printing the result  
17    Ada.Text_IO.Put_Line ("D =" & D'Img);  
18 end Main;
```

Build output

```
main.adb:6:07: warning: "B" is not modified, could be declared constant [-gnatwk]  
main.adb:7:04: warning: "C" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
D = 101
```

You'll notice that, in both languages, statements are terminated with a semicolon. This means that you can have multi-line statements.

The shortcuts of incrementing and decrementing

You may have noticed that Ada does not have something similar to the `a++` or `a--` operators. Instead you must use the full assignment `A := A + 1` or `A := A - 1`.

In the Ada example above, there are two distinct sections to the `procedure Main`. This first section is delimited by the `is` keyword and the `begin` keyword. This section is called the declarative block of the subprogram. The declarative block is where you will define all the local variables which will be used in the subprogram. C89 had something similar, where developers were required to declare their variables at the top of the scope block. Most C developers may have run into this before when trying to write a for loop:

[C]

Listing 8: main.c

```
1 /* The C89 version */  
2
```

(continues on next page)

(continued from previous page)

```

3 #include <stdio.h>
4
5 int average(int* list, int length)
6 {
7     int i;
8     int sum = 0;
9
10    for(i = 0; i < length; ++i) {
11        sum += list[i];
12    }
13    return (sum / length);
14}
15
16 int main(int argc, const char * argv[])
17 {
18     int vals[] = { 2, 2, 4, 4 };
19
20     printf("Average: %d\n", average(vals, 4));
21
22     return 0;
23 }
```

Runtime output

Average: 3

[C]

Listing 9: main.c

```

1 // The modern C way
2
3 #include <stdio.h>
4
5 int average(int* list, int length)
6 {
7     int sum = 0;
8
9     for(int i = 0; i < length; ++i) {
10        sum += list[i];
11    }
12
13    return (sum / length);
14}
15
16 int main(int argc, const char * argv[])
17 {
18     int vals[] = { 2, 2, 4, 4 };
19
20     printf("Average: %d\n", average(vals, 4));
21
22     return 0;
23 }
```

Runtime output

Average: 3

For the fun of it, let's also see the Ada way to do this:

[Ada]

Listing 10: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main is
4     type Int_Array is array (Natural range <>) of Integer;
5
6     function Average (List : Int_Array) return Integer
7     is
8         Sum : Integer := 0;
9     begin
10        for I in List'Range loop
11            Sum := Sum + List (I);
12        end loop;
13
14        return (Sum / List'Length);
15    end Average;
16
17    Vals : constant Int_Array (1 .. 4) := (2, 2, 4, 4);
18 begin
19     Ada.Text_IO.Put_Line ("Average: " & Integer'Image (Average (Vals)));
20 end Main;
```

Runtime output

```
Average: 3
```

We will explore more about the syntax of loops in Ada in a future section of this course; but for now, notice that the `I` variable used as the loop index is not declared in the declarative section!

Declaration Flippy Floppy

Something peculiar that you may have noticed about declarations in Ada is that they are backwards from the way C does declarations. The C language expects the type followed by the variable name. Ada expects the variable name followed by a semicolon and then the type.

The next block in the Ada example is between the `begin` and `end` keywords. This is where your statements will live. You can create new scopes by using the `declare` keyword:

[Ada]

Listing 11: main.adb

```
1 with Ada.Text_IO;
2
3 procedure Main
4 is
5     -- variable declaration
6     A, B : Integer := 0;
7     C : Integer := 100;
8     D : Integer;
9 begin
10    -- Ada does not have a shortcut format for increment like in C
11    A := A + 1;
12
13    -- regular addition
14    D := A + B + C;
```

(continues on next page)

(continued from previous page)

```

16  -- printing the result
17  Ada.Text_IO.Put_Line ("D =" & D'Img);
18
19  declare
20      E : constant Integer := D * 100;
21  begin
22      -- printing the result
23      Ada.Text_IO.Put_Line ("E =" & E'Img);
24  end;
25
26 end Main;

```

Build output

```
main.adb:6:07: warning: "B" is not modified, could be declared constant [-gnatwk]
main.adb:7:04: warning: "C" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
D = 101
E = 10100
```

Notice that we declared a new variable E whose scope only exists in our newly defined block. The equivalent C code is:

[C]

Listing 12: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     // variable declarations
6     int a = 0, b = 0, c = 100, d;
7
8     // c shorthand for increment
9     a++;
10
11    // regular addition
12    d = a + b + c;
13
14    // printing the result
15    printf("d = %d\n", d);
16
17    {
18        const int e = d * 100;
19        printf("e = %d\n", e);
20    }
21
22    return 0;
23 }
```

Runtime output

```
d = 101
e = 10100
```

Fun Fact about the C language assignment operator =: Did you know that an assignment in C can be used in an expression? Let's look at an example:

[C]

Listing 13: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int a = 0;
6
7     if (a = 10)
8         printf("True\n");
9     else
10        printf("False\n");
11
12    return 0;
13 }
```

Runtime output

```
True
```

Run the above code example. What does it output? Is that what you were expecting?

The author of the above code example probably meant to test if `a == 10` in the if statement but accidentally typed `=` instead of `==`. Because C treats assignment as an expression, it was able to evaluate `a = 10`.

Let's look at the equivalent Ada code:

[Ada]

Listing 14: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main
4 is
5     A : Integer := 0;
6 begin
7
8     if A := 10 then
9         Put_Line ("True");
10    else
11        Put_Line ("False");
12    end if;
13 end Main;
```

The above code will not compile. This is because Ada does not allow assignment as an expression.

The "use" clause

You'll notice in the above code example, after `with Ada.Text_IO;` there is a new statement we haven't seen before — `use Ada.Text_IO;`. You may also notice that we are not using the `Ada.Text_IO` prefix before the `Put_Line` statements. When we add the `use` clause it tells the compiler that we won't be using the prefix in the call to subprograms of that package. The `use` clause is something to use with caution. For example: if we use the `Ada.Text_IO` package and we also have a `Put_Line` subprogram in our current compilation unit with the same signature, we have a (potential) collision!

63.9 Conditions

The syntax of an if statement:

[C]

Listing 15: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     // try changing the initial value to change the
6     // output of the program
7     int v = 0;
8
9     if (v > 0) {
10         printf("Positive\n");
11     }
12     else if (v < 0) {
13         printf("Negative\n");
14     }
15     else {
16         printf("Zero\n");
17     }
18
19     return 0;
20 }
```

Runtime output

Zero

[Ada]

Listing 16: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main
4 is
5     -- try changing the initial value to change the
6     -- output of the program
7     V : constant Integer := 0;
8 begin
9     if V > 0 then
10         Put_Line ("Positive");
11     elsif V < 0 then
12         Put_Line ("Negative");
13     else
14         Put_Line ("Zero");
15     end if;
16 end Main;
```

Build output

```
main.adb:9:09: warning: condition is always False [-gnatwc]
main.adb:11:12: warning: condition is always False [-gnatwc]
```

Runtime output

Zero

In Ada, everything that appears between the **if** and **then** keywords is the conditional expression, no parentheses are required. Comparison operators are the same except for:

Operator	C	Ada
Equality	<code>==</code>	<code>=</code>
Inequality	<code>!=</code>	<code>/=</code>
Not	<code>!</code>	not
And	<code>&&</code>	and
Or	<code> </code>	or

The syntax of a switch/case statement:

[C]

Listing 17: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     // try changing the initial value to change the
6     // output of the program
7     int v = 0;
8
9     switch(v) {
10         case 0:
11             printf("Zero\n");
12             break;
13         case 1: case 2: case 3: case 4: case 5:
14         case 6: case 7: case 8: case 9:
15             printf("Positive\n");
16             break;
17         case 10: case 12: case 14: case 16: case 18:
18             printf("Even number between 10 and 18\n");
19             break;
20         default:
21             printf("Something else\n");
22             break;
23     }
24
25     return 0;
26 }
```

Runtime output

Zero

[Ada]

Listing 18: main.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Main
4 is
5     -- try changing the initial value to change the
6     -- output of the program
7     V : constant Integer := 0;
8 begin
```

(continues on next page)

(continued from previous page)

```

9  case V is
10 when 0 =>
11     Put_Line ("Zero");
12 when 1 .. 9 =>
13     Put_Line ("Positive");
14 when 10 | 12 | 14 | 16 | 18 =>
15     Put_Line ("Even number between 10 and 18");
16 when others =>
17     Put_Line ("Something else");
18 end case;
19 end Main;

```

Runtime output

Zero

Switch or Case?

A switch statement in C is the same as a case statement in Ada. This may be a little strange because C uses both keywords in the statement syntax. Let's make an analogy between C and Ada: C's **switch** is to Ada's **case** as C's **case** is to Ada's **when**.

Notice that in Ada, the case statement does not use the **break** keyword. In C, we use **break** to stop the execution of a case branch from falling through to the next branch. Here is an example:

[C]

Listing 19: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int v = 0;
6
7     switch(v) {
8         case 0:
9             printf("Zero\n");
10        case 1:
11            printf("One\n");
12        default:
13            printf("Other\n");
14    }
15
16    return 0;
17 }

```

Runtime output

```

Zero
One
Other

```

Run the above code with `v = 0`. What prints? What prints when we change the assignment to `v = 1`?

When `v = 0` the program outputs the strings Zero then One then Other. This is called fall through. If you add the **break** statements back into the **switch** you can stop this fall through behavior from happening. The reason why fall through is allowed in C is to allow

the behavior from the previous example where we want a specific branch to execute for multiple inputs. Ada solves this a different way because it is possible, or even probable, that the developer might forget a **break** statement accidentally. So Ada does not allow fall through. Instead, you can use Ada's syntax to identify when a specific branch can be executed by more than one input. If you want a range of values for a specific branch you can use the `First .. Last` notation. If you want a few non-consecutive values you can use the `Value1 | Value2 | Value3` notation.

Instead of using the word **default** to denote the catch-all case, Ada uses the **others** keyword.

63.10 Loops

Let's start with some syntax:

[C]

Listing 20: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int v;
6
7     // this is a while loop
8     v = 1;
9     while(v < 100) {
10         v *= 2;
11     }
12     printf("v = %d\n", v);
13
14     // this is a do while loop
15     v = 1;
16     do {
17         v *= 2;
18     } while(v < 200);
19     printf("v = %d\n", v);
20
21     // this is a for loop
22     v = 0;
23     for(int i = 0; i < 5; ++i) {
24         v += (i * i);
25     }
26     printf("v = %d\n", v);
27
28     // this is a forever loop with a conditional exit
29     v = 0;
30     while(1) {
31         // do stuff here
32         v += 1;
33         if(v == 10)
34             break;
35     }
36     printf("v = %d\n", v);
37
38     // this is a loop over an array
39     {
40         #define ARR_SIZE (10)
41         const int arr[ARR_SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

```

(continues on next page)

(continued from previous page)

```

42     int sum = 0;
43
44     for(int i = 0; i < ARR_SIZE; ++i) {
45         sum += arr[i];
46     }
47     printf("sum = %d\n", sum);
48 }
49
50     return 0;
51 }
```

Runtime output

```
v = 128
v = 256
v = 30
v = 10
sum = 55
```

[Ada]

Listing 21: main.adb

```

1  with Ada.Text_Io;
2
3  procedure Main is
4      V : Integer;
5  begin
6      -- this is a while loop
7      V := 1;
8      while V < 100 loop
9          V := V * 2;
10     end loop;
11     Ada.Text_Io.Put_Line ("V = " & Integer'Image (V));
12
13     -- Ada doesn't have an explicit do while loop
14     -- instead you can use the loop and exit keywords
15     V := 1;
16     loop
17         V := V * 2;
18         exit when V >= 200;
19     end loop;
20     Ada.Text_Io.Put_Line ("V = " & Integer'Image (V));
21
22     -- this is a for loop
23     V := 0;
24     for I in 0 .. 4 loop
25         V := V + (I * I);
26     end loop;
27     Ada.Text_Io.Put_Line ("V = " & Integer'Image (V));
28
29     -- this is a forever loop with a conditional exit
30     V := 0;
31     loop
32         -- do stuff here
33         V := V + 1;
34         exit when V = 10;
35     end loop;
36     Ada.Text_Io.Put_Line ("V = " & Integer'Image (V));
37
38     -- this is a loop over an array
```

(continues on next page)

(continued from previous page)

```

39 declare
40     type Int_Array is array (Natural range 1 .. 10) of Integer;
41
42     Arr : constant Int_Array := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
43     Sum : Integer := 0;
44 begin
45     for I in Arr'Range loop
46         Sum := Sum + Arr (I);
47     end loop;
48     Ada.Text_Io.Put_Line ("Sum = " & Integer'Image (Sum));
49 end;
50 end Main;

```

Runtime output

```

V = 128
V = 256
V = 30
V = 10
Sum = 55

```

The loop syntax in Ada is pretty straightforward. The `loop` and `end loop` keywords are used to open and close the loop scope. Instead of using the `break` keyword to exit the loop, Ada has the `exit` statement. The `exit` statement can be combined with a logic expression using the `exit when` syntax.

The major deviation in loop syntax is regarding for loops. You'll notice, in C, that you sometimes declare, and at least initialize a loop counter variable, specify a loop predicate, or an expression that indicates when the loop should continue executing or complete, and last you specify an expression to update the loop counter.

[C]

```

for (initialization expression; loop predicate; update expression) {
    // some statements
}

```

In Ada, you don't declare or initialize a loop counter or specify an update expression. You only name the loop counter and give it a range to loop over. The loop counter is **read-only!** You cannot modify the loop counter inside the loop like you can in C. And the loop counter will increment consecutively along the specified range. But what if you want to loop over the range in reverse order?

[C]

Listing 22: main.c

```

1 #include <stdio.h>
2
3 #define MY_RANGE (10)
4
5 int main(int argc, const char * argv[])
6 {
7
8     for (int i = MY_RANGE; i >= 0; --i) {
9         printf("%d\n", i);
10    }
11
12    return 0;
13 }

```

Runtime output

```

10
9
8
7
6
5
4
3
2
1
0

```

[Ada]

Listing 23: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main
4  is
5      My_Range : constant := 10;
6  begin
7      for I in reverse 0 .. My_Range loop
8          Put_Line (I'Img);
9      end loop;
10 end Main;

```

Runtime output

```

10
9
8
7
6
5
4
3
2
1
0

```

Tick Image

Strangely enough, Ada people call the single apostrophe symbol, ', "tick". This "tick" says the we are accessing an attribute of the variable. When we do `I'Img` on a variable of a numerical type, we are going to return the string version of that numerical type. So in the for loop above, `I'Img`, or "I tick image" will return the string representation of the numerical value stored in `I`. We have to do this because `Put_Line` is expecting a string as an input parameter.

We'll discuss attributes in more details *later in this chapter* (page 664).

In the above example, we are traversing over the range in reverse order. In Ada, we use the `reverse` keyword to accomplish this.

In many cases, when we are writing a for loop, it has something to do with traversing an array. In C, this is a classic location for off-by-one errors. Let's see an example in action:

[C]

Listing 24: main.c

```
1 #include <stdio.h>
2
3 #define LIST_LENGTH (100)
4
5 int main(int argc, const char * argv[])
6 {
7     int list[LIST_LENGTH];
8
9     for(int i = LIST_LENGTH; i > 0; --i) {
10         list[i] = LIST_LENGTH - i;
11     }
12
13     for (int i = 0; i < LIST_LENGTH; ++i)
14     {
15         printf("%d ", list[i]);
16
17         if (i % 10 == 0) {
18             printf("\n");
19         }
20     }
21
22     return 0;
23 }
```

Runtime output

```
0
99 98 97 96 95 94 93 92 91 90
89 88 87 86 85 84 83 82 81 80
79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60
59 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40
39 38 37 36 35 34 33 32 31 30
29 28 27 26 25 24 23 22 21 20
19 18 17 16 15 14 13 12 11 10
9 8 7 6 5 4 3 2 1
```

[Ada]

Listing 25: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main
4 is
5     type Int_Array is array (Natural range 1 .. 100) of Integer;
6
7     List : Int_Array;
8 begin
9
10    for I in reverse List'Range loop
11        List (I) := List'Last - I;
12    end loop;
13
14    for I in List'Range loop
15        Put (List (I)'Img & " ");
16
17        if I mod 10 = 0 then
```

(continues on next page)

(continued from previous page)

```

18      New_Line;
19  end if;
20 end loop;
21
22 end Main;

```

Runtime output

99	98	97	96	95	94	93	92	91	90
89	88	87	86	85	84	83	82	81	80
79	78	77	76	75	74	73	72	71	70
69	68	67	66	65	64	63	62	61	60
59	58	57	56	55	54	53	52	51	50
49	48	47	46	45	44	43	42	41	40
39	38	37	36	35	34	33	32	31	30
29	28	27	26	25	24	23	22	21	20
19	18	17	16	15	14	13	12	11	10
9	8	7	6	5	4	3	2	1	0

The above Ada and C code should initialize an array using a for loop. The initial values in the array should be contiguously decreasing from 99 to 0 as we index from the first index to the last index. In other words, the first index has a value of 99, the next has 98, the next 97 ... the last has a value of 0.

If you run both the C and Ada code above you'll notice that the outputs of the two programs are different. Can you spot why?

In the C code there are two problems:

1. There's a buffer overflow in the first iteration of the loop. We would need to modify the loop initialization to `int i = LIST_LENGTH - 1;`. The loop predicate should be modified to `i >= 0;`
2. The C code also has another off-by-one problem in the math to compute the value stored in `list[i]`. The expression should be changed to be `list[i] = LIST_LENGTH - i - 1;`.

These are typical off-by-one problems that plagues C programs. You'll notice that we didn't have this problem with the Ada code because we aren't defining the loop with arbitrary numeric literals. Instead we are accessing attributes of the array we want to manipulate and are using a keyword to determine the indexing direction.

We can actually simplify the Ada for loop a little further using iterators:

[Ada]

Listing 26: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main
4  is
5    type Int_Array is array (Natural range 1 .. 100) of Integer;
6
7    List : Int_Array;
8  begin
9
10   for I in reverse List'Range loop
11     List (I) := List'Last - I;
12   end loop;
13
14   for I of List loop

```

(continues on next page)

(continued from previous page)

```

15    Put (I'Img & " ");
16
17    if I mod 10 = 0 then
18        New_Line;
19    end if;
20    end loop;
21
22 end Main;
```

Runtime output

99	98	97	96	95	94	93	92	91	90
89	88	87	86	85	84	83	82	81	80
79	78	77	76	75	74	73	72	71	70
69	68	67	66	65	64	63	62	61	60
59	58	57	56	55	54	53	52	51	50
49	48	47	46	45	44	43	42	41	40
39	38	37	36	35	34	33	32	31	30
29	28	27	26	25	24	23	22	21	20
19	18	17	16	15	14	13	12	11	10
9	8	7	6	5	4	3	2	1	0

In the second for loop, we changed the syntax to `for I of List`. Instead of I being the index counter, it is now an iterator that references the underlying element. This example of Ada code is identical to the last bit of Ada code. We just used a different method to index over the second for loop. There is no C equivalent to this Ada feature, but it is similar to C++'s range based for loop.

63.11 Type System

63.11.1 Strong Typing

Ada is considered a "strongly typed" language. This means that the language does not define any implicit type conversions. C does define implicit type conversions, sometimes referred to as *integer promotion*. The rules for promotion are fairly straightforward in simple expressions but can get confusing very quickly. Let's look at a typical place of confusion with implicit type conversion:

[C]

Listing 27: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     unsigned char a = 0xFF;
6     char b = 0xFF;
7
8     printf("Does a == b?\n");
9     if(a == b)
10         printf("Yes.\n");
11     else
12         printf("No.\n");
13
14     printf("a: 0x%08X, b: 0x%08X\n", a, b);
15 }
```

(continues on next page)

(continued from previous page)

```
16    return 0;
17 }
```

Runtime output

```
Does a == b?
No.
a: 0x000000FF, b: 0xFFFFFFFF
```

Run the above code. You will notice that `a != b!` If we look at the output of the last `printf` statement we will see the problem. `a` is an unsigned number where `b` is a signed number. We stored a value of `0xFF` in both variables, but `a` treated this as the decimal number [255](#) while `b` treated this as the decimal number [-1](#). When we compare the two variables, of course they aren't equal; but that's not very intuitive. Let's look at the equivalent Ada example:

[Ada]

Listing 28: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main
4 is
5   type Char is range 0 .. 255;
6   type Unsigned_Char is mod 256;
7
8   A : Char := 16#FF#;
9   B : Unsigned_Char := 16#FF#;
10 begin
11
12   Put_Line ("Does A = B?");
13
14   if A = B then
15     Put_Line ("Yes");
16   else
17     Put_Line ("No");
18   end if;
19
20 end Main;
```

Build output

```
main.adb:14:09: error: invalid operand types for operator "="
main.adb:14:09: error: left operand has type "Char" defined at line 5
main.adb:14:09: error: right operand has type "Unsigned_Char" defined at line 6
gprbuild: *** compilation phase failed
```

If you try to run this Ada example you will get a compilation error. This is because the compiler is telling you that you cannot compare variables of two different types. We would need to explicitly cast one side to make the comparison against two variables of the same type. By enforcing the explicit cast we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect.

Another example: you can't divide an integer by a float. You need to perform the division operation using values of the same type, so one value must be explicitly converted to match the type of the other (in this case the more likely conversion is from integer to float). Ada is designed to guarantee that what's done by the program is what's meant by the programmer, leaving as little room for compiler interpretation as possible. Let's have a look at the following example:

[Ada]

Listing 29: strong_typing.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Strong_Typing is
4     Alpha : constant Integer := 1;
5     Beta  : constant Integer := 10;
6     Result : Float;
7 begin
8     Result := Float(Alpha) / Float(Beta);
9
10    Put_Line(Float'Image(Result));
11 end Strong_Typing;
```

Runtime output

```
1.00000E-01
```

[C]

Listing 30: main.c

```
1 #include <stdio.h>
2
3 void weakTyping (void) {
4     const int alpha = 1;
5     const int beta = 10;
6     float result;
7
8     result = alpha / beta;
9
10    printf("%f\n", result);
11 }
12
13 int main(int argc, const char * argv[])
14 {
15     weakTyping();
16
17     return 0;
18 }
```

Runtime output

```
0.000000
```

Are the three programs above equivalent? It may seem like Ada is just adding extra complexity by forcing you to make the conversion from **Integer** to **Float** explicit. In fact, it significantly changes the behavior of the computation. While the Ada code performs a floating point operation $1.0 / 10.0$ and stores 0.1 in **Result**, the C version instead store 0.0 in **result**. This is because the C version perform an integer operation between two integer variables: $1 / 10$ is 0. The result of the integer division is then converted to a **float** and stored. Errors of this sort can be very hard to locate in complex pieces of code, and systematic specification of how the operation should be interpreted helps to avoid this class of errors. If an integer division was actually intended in the Ada case, it is still necessary to explicitly convert the final result to **Float**:

[Ada]

```
-- Perform an Integer division then convert to Float
Result := Float(Alpha / Beta);
```

The complete example would then be:

[Ada]

Listing 31: strong_typing.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Strong_Typing is
4      Alpha : constant Integer := 1;
5      Beta  : constant Integer := 10;
6      Result : Float;
7  begin
8      Result := Float (Alpha / Beta);
9
10     Put_Line (Float'Image (Result));
11 end Strong_Typing;
```

Runtime output

0.00000E+00

Floating Point Literals

In Ada, a floating point literal must be written with both an integral and decimal part. `10` is not a valid literal for a floating point value, while `10.0` is.

63.11.2 Language-Defined Types

The principal scalar types predefined by Ada are `Integer`, `Float`, `Boolean`, and `Character`. These correspond to `int`, `float`, `int` (when used for Booleans), and `char`, respectively. The names for these types are not reserved words; they are regular identifiers. There are other language-defined integer and floating-point types as well. All have implementation-defined ranges and precision.

63.11.3 Application-Defined Types

Ada's type system encourages programmers to think about data at a high level of abstraction. The compiler will at times output a simple efficient machine instruction for a full line of source code (and some instructions can be eliminated entirely). The careful programmer's concern that the operation really makes sense in the real world would be satisfied, and so would the programmer's concern about performance.

The next example below defines two different metrics: area and distance. Mixing these two metrics must be done with great care, as certain operations do not make sense, like adding an area to a distance. Others require knowledge of the expected semantics; for example, multiplying two distances. To help avoid errors, Ada requires that each of the binary operators `+`, `-`, `*`, and `/` for integer and floating-point types take operands of the same type and return a value of that type.

[Ada]

Listing 32: main.adb

```

1  procedure Main is
2      type Distance is new Float;
3      type Area is new Float;
```

(continues on next page)

(continued from previous page)

```

5   D1 : Distance := 2.0;
6   D2 : Distance := 3.0;
7   A : Area;
8 begin
9     D1 := D1 + D2; -- OK
10    D1 := D1 + A; -- NOT OK: incompatible types for "+"
11    A := D1 * D2; -- NOT OK: incompatible types for ":="
12    A := Area (D1 * D2); -- OK
13 end Main;

```

Build output

```

main.adb:10:13: error: invalid operand types for operator "+"
main.adb:10:13: error: left operand has type "Distance" defined at line 2
main.adb:10:13: error: right operand has type "Area" defined at line 3
main.adb:11:04: warning: useless assignment to "A", value overwritten at line 12 [-gnatwm]
main.adb:11:13: error: expected type "Area" defined at line 3
main.adb:11:13: error: found type "Distance" defined at line 2
main.adb:12:04: warning: possibly useless assignment to "A", value might not be referenced [-gnatwm]
gprbuild: *** compilation phase failed

```

Even though the `Distance` and `Area` types above are just **Float**, the compiler does not allow arbitrary mixing of values of these different types. An explicit conversion (which does not necessarily mean any additional object code) is necessary.

The predefined Ada rules are not perfect; they admit some problematic cases (for example multiplying two `Distance` yields a `Distance`) and prohibit some useful cases (for example multiplying two `Distances` should deliver an `Area`). These situations can be handled through other mechanisms. A predefined operation can be identified as abstract to make it unavailable; overloading can be used to give new interpretations to existing operator symbols, for example allowing an operator to return a value from a type different from its operands; and more generally, GNAT has introduced a facility that helps perform dimensionality checking.

Ada enumerations work similarly to C `enum`:

[Ada]

Listing 33: main.adb

```

1 procedure Main is
2   type Day is
3     (Monday,
4      Tuesday,
5      Wednesday,
6      Thursday,
7      Friday,
8      Saturday,
9      Sunday);
10
11   D : Day := Monday;
12 begin
13   null;
14 end Main;

```

Build output

```

main.adb:4:07: warning: literal "Tuesday" is not referenced [-gnatwu]
main.adb:5:07: warning: literal "Wednesday" is not referenced [-gnatwu]

```

(continues on next page)

(continued from previous page)

```
main.adb:6:07: warning: literal "Thursday" is not referenced [-gnatwu]
main.adb:7:07: warning: literal "Friday" is not referenced [-gnatwu]
main.adb:8:07: warning: literal "Saturday" is not referenced [-gnatwu]
main.adb:9:07: warning: literal "Sunday" is not referenced [-gnatwu]
main.adb:11:04: warning: variable "D" is not referenced [-gnatwu]
```

[C]

Listing 34: main.c

```
1 enum Day {
2     Monday,
3     Tuesday,
4     Wednesday,
5     Thursday,
6     Friday,
7     Saturday,
8     Sunday
9 };
10
11 int main(int argc, const char * argv[])
12 {
13     enum Day d = Monday;
14
15     return 0;
16 }
```

But even though such enumerations may be implemented by the compiler as numeric values, at the language level Ada will not confuse the fact that `Monday` is a `Day` and is not an **Integer**. You can compare a `Day` with another `Day`, though. To specify implementation details like the numeric values that correspond with enumeration values in C you include them in the original `enum` declaration:

[C]

Listing 35: main.c

```
1 #include <stdio.h>
2
3 enum Day {
4     Monday    = 10,
5     Tuesday   = 11,
6     Wednesday = 12,
7     Thursday  = 13,
8     Friday    = 14,
9     Saturday  = 15,
10    Sunday    = 16
11 };
12
13 int main(int argc, const char * argv[])
14 {
15     enum Day d = Monday;
16
17     printf("d = %d\n", d);
18
19     return 0;
20 }
```

Runtime output

```
d = 10
```

But in Ada you must use both a type definition for Day as well as a separate representation clause for it like:

[Ada]

Listing 36: main.adb

```
1  with Ada.Text_IO;
2
3  procedure Main is
4      type Day is
5          (Monday,
6              Tuesday,
7                  Wednesday,
8                      Thursday,
9                          Friday,
10                             Saturday,
11                                 Sunday);
12
13  -- Representation clause for Day type:
14  for Day use
15      (Monday    => 10,
16          Tuesday   => 11,
17              Wednesday => 12,
18                  Thursday  => 13,
19                      Friday    => 14,
20                          Saturday  => 15,
21                              Sunday    => 16);
22
23  D : Day := Monday;
24  V : Integer;
25 begin
26     V := Day'Enum_Rep (D);
27     Ada.Text_IO.Put_Line (Integer'Image (V));
28 end Main;
```

Build output

```
main.adb:23:04: warning: "D" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
10
```

Note that however, unlike C, values for enumerations in Ada have to be unique.

63.11.4 Type Ranges

Contracts can be associated with types and variables, to refine values and define what are considered valid values. The most common kind of contract is a *range constraint* introduced with the **range** reserved word, for example:

[Ada]

Listing 37: main.adb

```
1  procedure Main is
2      type Grade is range 0 .. 100;
3
4      G1, G2  : Grade;
5      N       : Integer;
```

(continues on next page)

(continued from previous page)

```

6 begin
7   -- ...
8   G1 := 80;          -- Initialization of N
9   G1 := N;           -- OK
10  G1 := Grade (N);    -- Illegal (type mismatch)
11  G2 := G1 + 10;      -- Legal, run-time range check
12  G1 := (G1 + G2) / 2; -- Legal, run-time range check
13 end Main;

```

Build output

```

main.adb:8:04: warning: useless assignment to "G1", value overwritten at line 9 [-gnatwm]
main.adb:9:04: warning: useless assignment to "G1", value overwritten at line 10 [-gnatwm]
main.adb:9:10: error: expected type "Grade" defined at line 2
main.adb:9:10: error: found type "Standard.Integer"
main.adb:12:04: warning: possibly useless assignment to "G1", value might not be referenced [-gnatwm]
gprbuild: *** compilation phase failed

```

In the above example, Grade is a new integer type associated with a range check. Range checks are dynamic and are meant to enforce the property that no object of the given type can have a value outside the specified range. In this example, the first assignment to G1 is correct and will not raise a run-time exception. Assigning N to G1 is illegal since Grade is a different type than **Integer**. Converting N to Grade makes the assignment legal, and a range check on the conversion confirms that the value is within 0 .. 100. Assigning G1 + 10 to G2 is legal since + for Grade returns a Grade (note that the literal 10 is interpreted as a Grade value in this context), and again there is a range check.

The final assignment illustrates an interesting but subtle point. The subexpression G1 + G2 may be outside the range of Grade, but the final result will be in range. Nevertheless, depending on the representation chosen for Grade, the addition may overflow. If the compiler represents Grade values as signed 8-bit integers (i.e., machine numbers in the range -128 .. 127) then the sum G1 + G2 may exceed 127, resulting in an integer overflow. To prevent this, you can use explicit conversions and perform the computation in a sufficiently large integer type, for example:

[Ada]

Listing 38: main.adb

```

1 with Ada.Text_IO;
2
3 procedure Main is
4   type Grade is range 0 .. 100;
5
6   G1, G2 : Grade := 99;
7 begin
8   G1 := Grade ((Integer (G1) + Integer (G2)) / 2);
9   Ada.Text_IO.Put_Line (Grade'Image (G1));
10  end Main;

```

Build output

```

main.adb:6:08: warning: "G2" is not modified, could be declared constant [-gnatwk]

```

Runtime output

99

Range checks are useful for detecting errors as early as possible. However, there may be some impact on performance. Modern compilers do know how to remove redundant checks, and you can deactivate these checks altogether if you have sufficient confidence that your code will function correctly.

Types can be derived from the representation of any other type. The new derived type can be associated with new constraints and operations. Going back to the Day example, one can write:

[Ada]

Listing 39: main.adb

```
1 procedure Main is
2   type Day is
3     (Monday,
4      Tuesday,
5      Wednesday,
6      Thursday,
7      Friday,
8      Saturday,
9      Sunday);
10
11  type Business_Day is new Day range Monday .. Friday;
12  type Weekend_Day is new Day range Saturday .. Sunday;
13 begin
14   null;
15 end Main;
```

Build output

```
main.adb:4:07: warning: literal "Tuesday" is not referenced [-gnatwu]
main.adb:5:07: warning: literal "Wednesday" is not referenced [-gnatwu]
main.adb:6:07: warning: literal "Thursday" is not referenced [-gnatwu]
main.adb:11:09: warning: type "Business_Day" is not referenced [-gnatwu]
main.adb:12:09: warning: type "Weekend_Day" is not referenced [-gnatwu]
```

Since these are new types, implicit conversions are not allowed. In this case, it's more natural to create a new set of constraints for the same type, instead of making completely new ones. This is the idea behind *subtypes* in Ada. A subtype is a type with optional additional constraints. For example:

[Ada]

Listing 40: main.adb

```
1 procedure Main is
2   type Day is
3     (Monday,
4      Tuesday,
5      Wednesday,
6      Thursday,
7      Friday,
8      Saturday,
9      Sunday);
10
11  subtype Business_Day is Day range Monday .. Friday;
12  subtype Weekend_Day is Day range Saturday .. Sunday;
13  subtype Dice_Throw is Integer range 1 .. 6;
14 begin
15   null;
16 end Main;
```

Build output

```
main.adb:4:07: warning: literal "Tuesday" is not referenced [-gnatwu]
main.adb:5:07: warning: literal "Wednesday" is not referenced [-gnatwu]
main.adb:6:07: warning: literal "Thursday" is not referenced [-gnatwu]
main.adb:11:12: warning: type "Business_Day" is not referenced [-gnatwu]
main.adb:12:12: warning: type "Weekend_Day" is not referenced [-gnatwu]
main.adb:13:12: warning: type "Dice_Throw" is not referenced [-gnatwu]
```

These declarations don't create new types, just new names for constrained ranges of their base types.

The purpose of numeric ranges is to express some application-specific constraint that we want the compiler to help us enforce. More importantly, we want the compiler to tell us when that constraint cannot be met — when the underlying hardware cannot support the range given. There are two things to consider:

- just a range constraint, such as `A : Integer range 0 .. 10;`, or
- a type declaration, such as `type Result is range 0 .. 1_000_000_000;`.

Both represent some sort of application-specific constraint, but in addition, the type declaration promotes portability because it won't compile on targets that do not have a sufficiently large hardware numeric type. That's a definition of portability that is preferable to having something compile anywhere but not run correctly, as in C.

63.11.5 Unsigned And Modular Types

Unsigned integer numbers are quite common in embedded applications. In C, you can use them by declaring `unsigned int` variables. In Ada, you have two options:

- declare custom *unsigned* range types;
 - In addition, you can declare custom range *subtypes* or use existing subtypes such as `Natural`.
- declare custom modular types.

The following table presents the main features of each type. We discuss these types right after.

Feature	[C] <code>unsigned int</code>	[Ada] Unsigned range	[Ada] Modular
Excludes negative value	✓	✓	✓
Wraparound	✓		✓

When declaring custom range types in Ada, you may use the full range in the same way as in C. For example, this is the declaration of a 32-bit unsigned integer type and the X variable in Ada:

[Ada]

Listing 41: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;
5
6   X : Unsigned_Int_32 := 42;
7 begin
8   Put_Line ("X = " & Unsigned_Int_32'Image (X));
9 end Main;
```

Build output

```
main.adb:6:04: warning: "X" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
X = 42
```

In C, when **unsigned int** has a size of 32 bits, this corresponds to the following declaration:

[C]

Listing 42: main.c

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 int main(int argc, const char * argv[])
5 {
6     unsigned int x = 42;
7     printf("x = %u\n", x);
8
9     return 0;
10 }
```

Runtime output

```
X = 42
```

Another strategy is to declare subtypes for existing signed types and specify just the range that excludes negative numbers. For example, let's declare a custom 32-bit signed type and its unsigned subtype:

[Ada]

Listing 43: main.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Main is
4     type Signed_Int_32 is range -2 ** 31 .. 2 ** 31 - 1;
5
6     subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. Signed_Int_32'Last;
7     -- Equivalent to:
8     -- subtype Unsigned_Int_31 is Signed_Int_32 range 0 .. 2 ** 31 - 1;
9
10    X : Unsigned_Int_31 := 42;
11 begin
12     Put_Line ("X = " & Unsigned_Int_31'Image (X));
13 end Main;
```

Build output

```
main.adb:10:04: warning: "X" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
X = 42
```

In this case, we're just skipping the sign bit of the **Signed_Int_32** type. In other words, while **Signed_Int_32** has a size of 32 bits, **Unsigned_Int_31** has a range of 31 bits, even if the base type has 32 bits.

Note that the declaration above is actually similar to the existing **Natural** subtype. Ada provides the following standard subtypes:

```
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

Since they're standard subtypes, you can declare variables of those subtypes directly in your implementation, in the same way as you can declare **Integer** variables.

As indicated in the table above, however, there is a difference in behavior for the variables we just declared, which occurs in case of overflow. Let's consider this C example:

[C]

Listing 44: main.c

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 int main(int argc, const char * argv[])
5 {
6     unsigned int x = UINT_MAX + 1;
7     /* Now: x == 0 */
8
9     printf("x = %u\n", x);
10
11    return 0;
12 }
```

Runtime output

```
x = 0
```

The corresponding code in Ada raises an exception:

[Ada]

Listing 45: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     type Unsigned_Int_32 is range 0 .. 2 ** 32 - 1;
5
6     X : Unsigned_Int_32 := Unsigned_Int_32'Last + 1;
7     -- Overflow: exception is raised!
8 begin
9     Put_Line ("X = " & Unsigned_Int_32'Image (X));
10 end Main;
```

Build output

```
main.adb:6:04: warning: "X" is not modified, could be declared constant [-gnatwk]
main.adb:6:48: warning: value not in range of type "Unsigned_Int_32" defined at
  line 4 [enabled by default]
main.adb:6:48: warning: Constraint_Error will be raised at run time [enabled by
  default]
```

Runtime output

```
raised CONSTRAINT_ERROR : main.adb:6 range check failed
```

While the C uses modulo arithmetic for unsigned integer, Ada doesn't use it for the `Unsigned_Int_32` type. Ada does, however, support modular types via type definitions using the `mod` keyword. In this example, we declare a 32-bit modular type:

[Ada]

Listing 46: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Unsigned_32 is mod 2**32;
5
6   X : Unsigned_32 := Unsigned_32'Last + 1;
7   -- Now: X = 0
8 begin
9   Put_Line ("X = " & Unsigned_32'Image (X));
10 end Main;
```

Build output

```
main.adb:6:04: warning: "X" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
X = 0
```

In this case, the behavior is the same as in the C declaration above.

Modular types, unlike Ada's signed integers, also provide bit-wise operations, a typical application for unsigned integers in C. In Ada, you can use operators such as `and`, `or`, `xor` and `not`. You can also use typical bit-shifting operations, such as `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left` and `Rotate_Right`.

63.11.6 Attributes

Attributes start with a single apostrophe ("tick"), and they allow you to query properties of, and perform certain actions on, declared entities such as types, objects, and subprograms. For example, you can determine the first and last bounds of scalar types, get the sizes of objects and types, and convert values to and from strings. This section provides an overview of how attributes work. For more information on the many attributes defined by the language, you can refer directly to the Ada Language Reference Manual.

The '`Image`' and '`Value`' attributes allow you to transform a scalar value into a `String` and vice-versa. For example:

[Ada]

Listing 47: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   A : Integer := 10;
5 begin
6   Put_Line (Integer'Image (A));
7   A := Integer'Value ("99");
8   Put_Line (Integer'Image (A));
9 end Main;
```

Runtime output

```
10
99
```

Important

Semantically, attributes are equivalent to subprograms. For example, `Integer'Image` is defined as follows:

```
function Integer'Image(Arg : Integer'Base) return String;
```

Certain attributes are provided only for certain kinds of types. For example, the '`Val`' and '`Pos`' attributes for an enumeration type associates a discrete value with its position among its peers. One circuitous way of moving to the next character of the ASCII table is:

[Ada]

Listing 48: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   C : Character := 'a';
5 begin
6   Put (C);
7   C := Character'Val (Character'Pos (C) + 1);
8   Put (C);
9 end Main;
```

Runtime output

```
ab
```

A more concise way to get the next value in Ada is to use the '`Succ`' attribute:

[Ada]

Listing 49: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   C : Character := 'a';
5 begin
6   Put (C);
7   C := Character'Succ (C);
8   Put (C);
9 end Main;
```

Runtime output

```
ab
```

You can get the previous value using the '`Pred`' attribute. Here is the equivalent in C:

[C]

Listing 50: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     char c = 'a';
6     printf("%c", c);
```

(continues on next page)

(continued from previous page)

```
7     C++;
8     printf("%c", c);
9
10    return 0;
11 }
```

Runtime output

```
ab
```

Other interesting examples are the '`First`' and '`Last`' attributes which, respectively, return the first and last values of a scalar type. Using 32-bit integers, for instance, `Integer'First` returns -2^{31} and `Integer'Last` returns $2^{31} - 1$.

63.11.7 Arrays and Strings

C arrays are pointers with offsets, but the same is not the case for Ada. Arrays in Ada are not interchangeable with operations on pointers, and array types are considered first-class citizens. They have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and you can specify both the starting and ending bounds — the lower bound doesn't necessarily have to be 0. Most of the time, array types need to be explicitly declared prior to the declaration of an object of that array type.

Here's an example of declaring an array of 26 characters, initializing the values from '`a`' to '`z`':

[Ada]

Listing 51: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Arr_Type is array (Integer range <>) of Character;
5      Arr : Arr_Type (1 .. 26);
6      C : Character := 'a';
7  begin
8      for I in Arr'Range loop
9          Arr (I) := C;
10         C := Character'Succ (C);
11
12         Put (Arr (I) & " ");
13
14         if I mod 7 = 0 then
15             New_Line;
16         end if;
17     end loop;
18 end Main;
```

Runtime output

```
a b c d e f g
h i j k l m n
o p q r s t u
v w x y z
```

[C]

Listing 52: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     char Arr [26];
6     char C = 'a';
7
8     for (int I = 0; I < 26; ++I) {
9         Arr [I] = C++;
10        printf ("%c ", Arr [I]);
11
12        if ((I + 1) % 7 == 0) {
13            printf ("\n");
14        }
15    }
16
17    return 0;
18 }
```

Runtime output

```
a b c d e f g
h i j k l m n
o p q r s t u
v w x y z
```

In C, only the size of the array is given during declaration. In Ada, array index ranges are specified using two values of a discrete type. In this example, the array type declaration specifies the use of **Integer** as the index type, but does not provide any constraints (use `<>`, pronounced *box*, to specify "no constraints"). The constraints are defined in the object declaration to be 1 to 26, inclusive. Arrays have an attribute called '**Range**'. In our example, `Arr'Range` can also be expressed as `Arr'First .. Arr'Last`; both expressions will resolve to `1 .. 26`. So the '**Range**' attribute supplies the bounds for our **for** loop. There is no risk of stating either of the bounds incorrectly, as one might do in C where `I <= 26` may be specified as the end-of-loop condition.

As in C, Ada **String** is an array of **Character**. Ada strings, importantly, are not delimited with the special character '`\0`' like they are in C. It is not necessary because Ada uses the array's bounds to determine where the string starts and stops.

Ada's predefined **String** type is very straightforward to use:

[Ada]

Listing 53: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     My_String : String (1 .. 19) := "This is an example!";
5 begin
6     Put_Line (My_String);
7 end Main;
```

Build output

```
main.adb:4:04: warning: "My_String" is not modified, could be declared constant [-gnatwk]
```

Runtime output

This is an example!

Unlike C, Ada does not offer escape sequences such as '`\n`'. Instead, explicit values from the ASCII package must be concatenated (via the concatenation operator, `&`). Here for example, is how to initialize a line of text ending with a new line:

[Ada]

Listing 54: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     My_String : String := "This is a line" & ASCII.LF;
5 begin
6     Put (My_String);
7 end Main;
```

Build output

```
main.adb:4:04: warning: "My_String" is not modified, could be declared constant [-gnatwk]
```

Runtime output

This is a line

You see here that no constraints are necessary for this variable definition. The initial value given allows the automatic determination of `My_String`'s bounds.

Ada offers high-level operations for copying, slicing, and assigning values to arrays. We'll start with assignment. In C, the assignment operator doesn't make a copy of the value of an array, but only copies the address or reference to the target variable. In Ada, the actual array contents are duplicated. To get the above behavior, actual pointer types would have to be defined and used.

[Ada]

Listing 55: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     type Arr_Type is array (Integer range <>) of Integer;
5     A1 : Arr_Type (1 .. 2);
6     A2 : Arr_Type (1 .. 2);
7 begin
8     A1 (1) := 0;
9     A1 (2) := 1;
10
11    A2 := A1;
12
13    for I in A2'Range loop
14        Put_Line (Integer'Image (A2 (I)));
15    end loop;
16 end Main;
```

Runtime output

0
1

[C]

Listing 56: main.c

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, const char * argv[])
5 {
6     int A1 [2];
7     int A2 [2];
8
9     A1 [0] = 0;
10    A1 [1] = 1;
11
12    memcpy (A2, A1, sizeof (int) * 2);
13
14    for (int i = 0; i < 2; i++) {
15        printf("%d\n", A2[i]);
16    }
17
18    return 0;
19 }
```

Runtime output

```

0
1
```

In all of the examples above, the source and destination arrays must have precisely the same number of elements. Ada allows you to easily specify a portion, or slice, of an array. So you can write the following:

[Ada]

Listing 57: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     type Arr_Type is array (Integer range <>) of Integer;
5     A1 : Arr_Type (1 .. 10) := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
6     A2 : Arr_Type (1 .. 5)  := (1, 2, 3, 4, 5);
7 begin
8     A2 (1 .. 3) := A1 (4 .. 6);
9
10    for I in A2'Range loop
11        Put_Line (Integer'Image (A2 (I)));
12    end loop;
13 end Main;
```

Build output

```
main.adb:5:04: warning: "A1" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```

4
5
6
4
5
```

This assigns the 4th, 5th, and 6th elements of A1 into the 1st, 2nd, and 3rd elements of A2. Note that only the length matters here: the values of the indexes don't have to be equal; they slide automatically.

Ada also offers high level comparison operations which compare the contents of arrays as opposed to their addresses:

[Ada]

Listing 58: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type Arr_Type is array (Integer range <>) of Integer;
5   A1 : Arr_Type (1 .. 2) := (10, 20);
6   A2 : Arr_Type (1 .. 2) := (10, 20);
7 begin
8   if A1 = A2 then
9     Put_Line ("A1 = A2");
10  else
11    Put_Line ("A1 /= A2");
12  end if;
13 end Main;
```

Build output

```
main.adb:5:04: warning: "A1" is not modified, could be declared constant [-gnatwk]
main.adb:6:04: warning: "A2" is not modified, could be declared constant [-gnatwk]
```

Runtime output

```
A1 = A2
```

[C]

Listing 59: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5   int A1 [2] = { 10, 20 };
6   int A2 [2] = { 10, 20 };
7
8   int eq = 1;
9
10  for (int i = 0; i < 2; ++i) {
11    if (A1 [i] != A2 [i]) {
12      eq = 0;
13      break;
14    }
15  }
16
17  if (eq) {
18    printf("A1 == A2\n");
19  }
20  else {
21    printf("A1 != A2\n");
22  }
23
24  return 0;
25 }
```

Runtime output

```
A1 == A2
```

You can assign to all the elements of an array in each language in different ways. In Ada, the number of elements to assign can be determined by looking at the right-hand side, the left-hand side, or both sides of the assignment. When bounds are known on the left-hand side, it's possible to use the others expression to define a default value for all the unspecified array elements. Therefore, you can write:

[Ada]

Listing 60: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Arr_Type is array (Integer range <>) of Integer;
5      A1 : Arr_Type (-2 .. 42) := (others => 0);
6  begin
7      -- use a slice to assign A1 elements 11 .. 19 to 1
8      A1 (11 .. 19) := (others => 1);
9
10     Put_Line ("---- A1 ----");
11     for I in A1'Range loop
12         Put_Line (Integer'Image (I) & " => " &
13                     Integer'Image (A1 (I)));
14     end loop;
15 end Main;
```

Runtime output

```
---- A1 ----
-2 => 0
-1 => 0
0 => 0
1 => 0
2 => 0
3 => 0
4 => 0
5 => 0
6 => 0
7 => 0
8 => 0
9 => 0
10 => 0
11 => 1
12 => 1
13 => 1
14 => 1
15 => 1
16 => 1
17 => 1
18 => 1
19 => 1
20 => 0
21 => 0
22 => 0
23 => 0
24 => 0
25 => 0
26 => 0
```

(continues on next page)

(continued from previous page)

```

27 => 0
28 => 0
29 => 0
30 => 0
31 => 0
32 => 0
33 => 0
34 => 0
35 => 0
36 => 0
37 => 0
38 => 0
39 => 0
40 => 0
41 => 0
42 => 0

```

In this example, we're specifying that A1 has a range between -2 and 42. We use (**others => 0**) to initialize all array elements with zero. In the next example, the number of elements is determined by looking at the right-hand side:

[Ada]

Listing 61: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type Arr_Type is array (Integer range <>) of Integer;
5      A1 : Arr_Type := (1, 2, 3, 4, 5, 6, 7, 8, 9);
6  begin
7      A1 := (1, 2, 3, others => 10);
8
9      Put_Line ("---- A1 ----");
10     for I in A1'Range loop
11         Put_Line (Integer'Image (I) & " => " &
12                     Integer'Image (A1 (I)));
13     end loop;
14 end Main;

```

Runtime output

```

---- A1 ----
-2147483648 => 1
-2147483647 => 2
-2147483646 => 3
-2147483645 => 10
-2147483644 => 10
-2147483643 => 10
-2147483642 => 10
-2147483641 => 10
-2147483640 => 10

```

Since A1 is initialized with an aggregate of 9 elements, A1 automatically has 9 elements. Also, we're not specifying any range in the declaration of A1. Therefore, the compiler uses the default range of the underlying array type Arr_Type, which has an unconstrained range based on the **Integer** type. The compiler selects the first element of that type (**Integer'First**) as the start index of A1. If you replaced **Integer range <>** in the declaration of the Arr_Type by **Positive range <>**, then A1's start index would be **Positive'First** — which corresponds to one.

63.11.8 Heterogeneous Data Structures

The structure corresponding to a C **struct** is an Ada **record**. Here are some simple records:

[Ada]

Listing 62: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Main is
4      type R is record
5          A, B : Integer;
6          C     : Float;
7      end record;
8
9      V : R;
10 begin
11     V.A := 0;
12     Put_Line ("V.A = " & Integer'Image (V.A));
13 end Main;
```

Runtime output

```
V.A = 0
```

[C]

Listing 63: main.c

```

1 #include <stdio.h>
2
3 struct R {
4     int A, B;
5     float C;
6 };
7
8 int main(int argc, const char * argv[])
9 {
10     struct R V;
11     V.A = 0;
12     printf("V.A = %d\n", V.A);
13
14     return 0;
15 }
```

Runtime output

```
V.A = 0
```

Ada allows specification of default values for fields just like C. The values specified can take the form of an ordered list of values, a named list of values, or an incomplete list followed by others => <> to specify that fields not listed will take their default values. For example:

[Ada]

Listing 64: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Main is
4
5      type R is record
```

(continues on next page)

(continued from previous page)

```

6   A, B : Integer := 0;
7   C     : Float   := 0.0;
8 end record;
9
10 procedure Put_R (V : R; Name : String) is
11 begin
12   Put_Line (Name & " = "
13             & Integer'Image (V.A) & ", "
14             & Integer'Image (V.B) & ", "
15             & Float'Image (V.C) & ")");
16 end Put_R;
17
18 V1 : constant R := (1, 2, 1.0);
19 V2 : constant R := (A => 1, B => 2, C => 1.0);
20 V3 : constant R := (C => 1.0, A => 1, B => 2);
21 V4 : constant R := (C => 1.0, others => <>);
22
23 begin
24   Put_R (V1, "V1");
25   Put_R (V2, "V2");
26   Put_R (V3, "V3");
27   Put_R (V4, "V4");
28 end Main;

```

Runtime output

```

V1 = ( 1, 2, 1.00000E+00)
V2 = ( 1, 2, 1.00000E+00)
V3 = ( 1, 2, 1.00000E+00)
V4 = ( 0, 0, 1.00000E+00)

```

63.11.9 Pointers

As a foreword to the topic of pointers, it's important to keep in mind the fact that most situations that would require a pointer in C do not in Ada. In the vast majority of cases, indirect memory management can be hidden from the developer and thus saves from many potential errors. However, there are situations that do require the use of pointers, or said differently that require to make memory indirection explicit. This section will present Ada access types, the equivalent of C pointers. A further section will provide more details as to how situations that require pointers in C can be done without access types in Ada.

We'll continue this section by explaining the difference between objects allocated on the stack and objects allocated on the heap using the following example:

[Ada]

Listing 65: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4   type R is record
5     A, B : Integer;
6   end record;
7
8   procedure Put_R (V : R; Name : String) is
9 begin
10   Put_Line (Name & " = "
11             & Integer'Image (V.A) & ", "

```

(continues on next page)

(continued from previous page)

```

12      & Integer'Image (V.B) & ")");
13 end Put_R;
14
15 V1, V2 : R;
16
17 begin
18   V1.A := 0;
19   V2 := V1;
20   V2.A := 1;
21
22   Put_R (V1, "V1");
23   Put_R (V2, "V2");
24 end Main;

```

Runtime output

```
V1 = ( 0,  0)
V2 = ( 1,  0)
```

[C]

Listing 66: main.c

```

1 #include <stdio.h>
2
3 struct R {
4   int A, B;
5 };
6
7 void print_r(const struct R *v,
8             const char    *name)
9 {
10   printf("%s = (%d, %d)\n", name, v->A, v->B);
11 }
12
13 int main(int argc, const char * argv[])
14 {
15   struct R V1, V2;
16   V1.A = 0;
17   V2 = V1;
18   V2.A = 1;
19
20   print_r(&V1, "V1");
21   print_r(&V2, "V2");
22
23   return 0;
24 }
```

Runtime output

```
V1 = (0, 0)
V2 = (1, 0)
```

There are many commonalities between the Ada and C semantics above. In Ada and C, objects are allocated on the stack and are directly accessed. V1 and V2 are two different objects and the assignment statement copies the value of V1 into V2. V1 and V2 are two distinct objects.

Here's now a similar example, but using heap allocation instead:

[Ada]

Listing 67: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4      type R is record
5          A, B : Integer;
6      end record;
7
8      type R_Access is access R;
9
10     procedure Put_R (V : R; Name : String) is
11    begin
12        Put_Line (Name & " = "
13                    & Integer'Image (V.A) & ", "
14                    & Integer'Image (V.B) & ")");
15    end Put_R;
16
17    V1 : R_Access;
18    V2 : R_Access;
19  begin
20      V1 := new R;
21      V1.A := 0;
22      V2 := V1;
23      V2.A := 1;
24
25      Put_R (V1.all, "V1");
26      Put_R (V2.all, "V2");
27  end Main;

```

Runtime output

```

V1 = ( 1,  0)
V2 = ( 1,  0)

```

[C]

Listing 68: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct R {
5     int A, B;
6 };
7
8 void print_r(const struct R *v,
9             const char *name)
{
10     printf("%s = (%d, %d)\n", name, v->A, v->B);
11 }
12
13 int main(int argc, const char * argv[])
14 {
15     struct R * V1, * V2;
16     V1 = malloc(sizeof(struct R));
17     V1->A = 0;
18     V2 = V1;
19     V2->A = 1;
20
21     print_r(V1, "V1");

```

(continues on next page)

(continued from previous page)

```

23     print_r(V2, "V2");
24
25     return 0;
26 }
```

Runtime output

```
V1 = (1, 0)
V2 = (1, 0)
```

In this example, an object of type R is allocated on the heap. The same object is then referred to through V1 and V2. As in C, there's no garbage collector in Ada, so objects allocated by the new operator need to be expressly freed (which is not the case here).

Dereferencing is performed automatically in certain situations, for instance when it is clear that the type required is the dereferenced object rather than the pointer itself, or when accessing record members via a pointer. To explicitly dereference an access variable, append `.all`. The equivalent of `V1->A` in C can be written either as `V1.A` or `V1.all.A`.

Pointers to scalar objects in Ada and C look like:

[Ada]

Listing 69: main.adb

```

1  procedure Main is
2      type A_Int is access Integer;
3      Var : A_Int := new Integer;
4
5  begin
6      Var.all := 0;
7  end Main;
```

Build output

```
main.adb:3:04: warning: "Var" is not modified, could be declared constant [-gnatwk]
```

[C]

Listing 70: main.c

```

1  #include <stdlib.h>
2
3  int main(int argc, const char * argv[])
4  {
5      int * Var = malloc (sizeof(int));
6      *Var = 0;
7      return 0;
8 }
```

In Ada, an initializer can be specified with the allocation by appending '`(value)`:

[Ada]

Listing 71: main.adb

```

1  procedure Main is
2      type A_Int is access Integer;
3
4      Var : A_Int := new Integer'(0);
5
6  begin
7      null;
8  end Main;
```

Build output

```
main.adb:4:04: warning: variable "Var" is not referenced [-gnatwu]
```

When using Ada pointers to reference objects on the stack, the referenced objects must be declared as being aliased. This directs the compiler to implement the object using a memory region, rather than using registers or eliminating it entirely via optimization. The access type needs to be declared as either **access all** (if the referenced object needs to be assigned to) or **access constant** (if the referenced object is a constant). The '**Access**' attribute works like the C & operator to get a pointer to the object, but with a *scope accessibility* check to prevent references to objects that have gone out of scope. For example:

[Ada]

Listing 72: main.adb

```
1 procedure Main is
2     type A_Int is access all Integer;
3     Var : aliased Integer;
4     Ptr : A_Int := Var'Access;
5 begin
6     null;
7 end Main;
```

Build output

```
main.adb:4:04: warning: variable "Ptr" is not referenced [-gnatwu]
```

[C]

Listing 73: main.c

```
1 int main(int argc, const char * argv[])
2 {
3     int Var;
4     int * Ptr = &Var;
5
6     return 0;
7 }
```

To deallocate objects from the heap in Ada, it is necessary to use a deallocation subprogram that accepts a specific access type. A generic procedure is provided that can be customized to fit your needs, it's called `Ada.Unchecked_Deallocation`. To create your customized deallocator (that is, to instantiate this generic), you must provide the object type as well as the access type as follows:

[Ada]

Listing 74: main.adb

```
1 with Ada.Unchecked_Deallocation;
2
3 procedure Main is
4     type Integer_Access is access all Integer;
5     procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);
6     My_Pointer : Integer_Access := new Integer;
7 begin
8     Free (My_Pointer);
9 end Main;
```

[C]

Listing 75: main.c

```

1 #include <stdlib.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int * my_pointer = malloc (sizeof(int));
6     free (my_pointer);
7
8     return 0;
9 }
```

We'll discuss generics later *in this section* (page 771).

63.12 Functions and Procedures

63.12.1 General Form

Subroutines in C are always expressed as functions which may or may not return a value. Ada explicitly differentiates between functions and procedures. Functions must return a value and procedures must not. Ada uses the more general term *subprogram* to refer to both functions and procedures.

Parameters can be passed in three distinct modes:

- **in**, which is the default, is for input parameters, whose value is provided by the caller and cannot be changed by the subprogram.
- **out** is for output parameters, with no initial value, to be assigned by the subprogram and returned to the caller.
- **in out** is a parameter with an initial value provided by the caller, which can be modified by the subprogram and returned to the caller (more or less the equivalent of a non-constant pointer in C).

Ada also provides **access** and **aliased** parameters, which are in effect explicit pass-by-reference indicators.

In Ada, the programmer specifies how the parameter will be used and in general the compiler decides how it will be passed (i.e., by copy or by reference). C has the programmer specify how to pass the parameter.

Important

There are some exceptions to the "general" rule in Ada. For example, parameters of scalar types are always passed by copy, for all three modes.

Here's a first example:

[Ada]

Listing 76: proc.ads

```

1 procedure Proc
2   (Var1 : Integer;
3    Var2 : out Integer;
4    Var3 : in out Integer);
```

Listing 77: func.ads

```
1 function Func (Var : Integer) return Integer;
```

Listing 78: proc.adb

```
1 with Func;
2
3 procedure Proc
4   (Var1 : Integer;
5    Var2 : out Integer;
6    Var3 : in out Integer)
7 is
8 begin
9   Var2 := Func (Var1);
10  Var3 := Var3 + 1;
11 end Proc;
```

Listing 79: func.adb

```
1 function Func (Var : Integer) return Integer
2 is
3 begin
4   return Var + 1;
5 end Func;
```

Listing 80: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Proc;
3
4 procedure Main is
5   V1, V2 : Integer;
6 begin
7   V2 := 2;
8   Proc (5, V1, V2);
9
10  Put_Line ("V1: " & Integer'Image (V1));
11  Put_Line ("V2: " & Integer'Image (V2));
12 end Main;
```

Runtime output

```
V1: 6
V2: 3
```

[C]

Listing 81: proc.h

```
1 void Proc
2   (int Var1,
3    int * Var2,
4    int * Var3);
```

Listing 82: func.h

```
1 int Func (int Var);
```

Listing 83: proc.c

```

1 #include "func.h"
2
3 void Proc
4     (int Var1,
5      int *Var2,
6      int *Var3)
7 {
8     *Var2 = Func (Var1);
9     *Var3 += 1;
10}

```

Listing 84: func.c

```

1 int Func (int Var)
2 {
3     return Var + 1;
4 }

```

Listing 85: main.c

```

1 #include <stdio.h>
2 #include "proc.h"
3
4 int main(int argc, const char * argv[])
5 {
6     int v1, v2;
7
8     v2 = 2;
9     Proc (5, &v1, &v2);
10
11    printf("v1: %d\n", v1);
12    printf("v2: %d\n", v2);
13
14    return 0;
15 }

```

Runtime output

```
v1: 6
v2: 3
```

The first two declarations for Proc and Func are specifications of the subprograms which are being provided later. Although optional here, it's still considered good practice to separately define specifications and implementations in order to make it easier to read the program. In Ada and C, a function that has not yet been seen cannot be used. Here, Proc can call Func because its specification has been declared.

Parameters in Ada subprogram declarations are separated with semicolons, because commas are reserved for listing multiple parameters of the same type. Parameter declaration syntax is the same as variable declaration syntax (except for the modes), including default values for parameters. If there are no parameters, the parentheses must be omitted entirely from both the declaration and invocation of the subprogram.

In Ada 202X

Ada 202X allows for using static expression functions, which are evaluated at compile time. To achieve this, we can use an aspect — we'll discuss aspects *later in this chapter* (page 684).

An expression function is static when the Static aspect is specified. For example:

```
procedure Main is

    X1 : constant := (if True then 37 else 42);

    function If_Then_Else (Flag : Boolean; X, Y : Integer)
        return Integer is
        (if Flag then X else Y) with Static;

    X2 : constant := If_Then_Else (True, 37, 42);

begin
    null;
end Main;
```

In this example, we declare X1 using an expression. In the declaration of X2, we call the static expression function If_Then_Else. Both X1 and X2 have the same constant value.

63.12.2 Overloading

In C, function names must be unique. Ada allows overloading, in which multiple subprograms can share the same name as long as the subprogram signatures (the parameter types, and function return types) are different. The compiler will be able to resolve the calls to the proper routines or it will reject the calls. For example:

[Ada]

Listing 86: machine.ads

```
1 package Machine is
2     type Status is (Off, On);
3     type Code is new Integer range 0 .. 3;
4     type Threshold is new Float range 0.0 .. 10.0;
5
6     function Get (S : Status) return Code;
7     function Get (S : Status) return Threshold;
8
9 end Machine;
```

Listing 87: machine.adb

```
1 package body Machine is
2
3     function Get (S : Status) return Code is
4 begin
5     case S is
6         when Off => return 1;
7         when On  => return 3;
8     end case;
9 end Get;
10
11    function Get (S : Status) return Threshold is
12 begin
13     case S is
14         when Off => return 2.0;
15         when On  => return 10.0;
16     end case;
17 end Get;
```

(continues on next page)

(continued from previous page)

```
18
19 end Machine;
```

Listing 88: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Machine;      use Machine;
3
4 procedure Main is
5   S : Status;
6   C : Code;
7   T : Threshold;
8 begin
9   S := On;
10  C := Get (S);
11  T := Get (S);
12
13  Put_Line ("S: " & Status'Image (S));
14  Put_Line ("C: " & Code'Image (C));
15  Put_Line ("T: " & Threshold'Image (T));
16 end Main;
```

Runtime output

```
S: ON
C: 3
T: 1.00000E+01
```

The Ada compiler knows that an assignment to C requires a Code value. So, it chooses the Get function that returns a Code to satisfy this requirement.

Operators in Ada are functions too. This allows you to define local operators that override operators defined at an outer scope, and provide overloaded operators that operate on and compare different types. To declare an operator as a function, enclose its "name" in quotes:

[Ada]

Listing 89: machine_2.ads

```
1 package Machine_2 is
2   type Status is (Off, Waiting, On);
3   type Input is new Float range 0.0 .. 10.0;
4
5   function Get (I : Input) return Status;
6
7   function "=" (Left : Input; Right : Status) return Boolean;
8
9 end Machine_2;
```

Listing 90: machine_2.adb

```
1 package body Machine_2 is
2
3   function Get (I : Input) return Status is
4 begin
5     if I >= 0.0 and I < 3.0 then
6       return Off;
7     elsif I >= 3.0 and I < 6.5 then
8       return Waiting;
9     else
10       return On;
```

(continues on next page)

(continued from previous page)

```

11    end if;
12 end Get;

13
14 function "=" (Left : Input; Right : Status) return Boolean is
15 begin
16     return Get (Left) = Right;
17 end "=";

18
19 end Machine_2;

```

Listing 91: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Machine_2;   use Machine_2;
3
4 procedure Main is
5   I : Input;
6 begin
7   I := 3.0;
8   if I = Off then
9     Put_Line ("Machine is off.");
10  else
11    Put_Line ("Machine is not off.");
12  end if;
13 end Main;

```

Runtime output

Machine is not off.

63.12.3 Aspects

Aspect specifications allow you to define certain characteristics of a declaration using the **with** keyword after the declaration:

```

procedure Some_Procedure is <procedure_definition>
  with Some_Aspect => <aspect_specification>;
  
function Some_Function is <function_definition>
  with Some_Aspect => <aspect_specification>;
  
type Some_Type is <type_definition>
  with Some_Aspect => <aspect_specification>;
  
Obj : Some_Type with Some_Aspect => <aspect_specification>;

```

For example, you can inline a subprogram by specifying the **Inline** aspect:

[Ada]

Listing 92: float_arrays.ads

```

1 package Float_Arrays is
2
3   type Float_Array is array (Positive range <>) of Float;
4
5   function Average (Data : Float_Array) return Float
6     with Inline;

```

(continues on next page)

(continued from previous page)

```
7 end Float_Arrays;
```

We'll discuss inlining *later in this course* (page 816).

Aspect specifications were introduced in Ada 2012. In previous versions of Ada, you had to use a **pragma** instead. The previous example would be written as follows:

[Ada]

Listing 93: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is array (Positive range <>) of Float;
4
5     function Average (Data : Float_Array) return Float;
6
7     pragma Inline (Average);
8
9 end Float_Arrays;
```

Aspects and attributes might refer to the same kind of information. For example, we can use the **Size** aspect to define the expected minimum size of objects of a certain type:

[Ada]

Listing 94: my_device_types.ads

```
1 package My_Device_Types is
2
3     type UInt10 is mod 2 ** 10
4         with Size => 10;
5
6 end My_Device_Types;
```

In the same way, we can use the **size** attribute to retrieve the size of a type or of an object:

[Ada]

Listing 95: show_device_types.adb

```
1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with My_Device_Types; use My_Device_Types;
4
5 procedure Show_Device_Types is
6     UInt10_Obj : constant UInt10 := 0;
7 begin
8     Put_Line ("Size of UInt10 type: " & Positive'Image (UInt10'Size));
9     Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_Obj'Size));
10 end Show_Device_Types;
```

We'll explain both **Size** aspect and **Size** attribute *later in this course* (page 714).

CONCURRENCY AND REAL-TIME

64.1 Understanding the various options

Concurrent and real-time programming are standard parts of the Ada language. As such, they have the same semantics, whether executing on a native target with an OS such as Linux, on a real-time operating system (RTOS) such as VxWorks, or on a bare metal target with no OS or RTOS at all.

For resource-constrained systems, two subsets of the Ada concurrency facilities are defined, known as the Ravenscar and Jorvik profiles. Though restricted, these subsets have highly desirable properties, including: efficiency, predictability, analyzability, absence of deadlock, bounded blocking, absence of priority inversion, a real-time scheduler, and a small memory footprint. On bare metal systems, this means in effect that Ada comes with its own real-time kernel.

For further information

We'll discuss the Ravenscar profile *later in this chapter* (page 697). Details about the Jorvik profile can be found elsewhere [[Jorvik](#)].

Enhanced portability and expressive power are the primary advantages of using the standard concurrency facilities, potentially resulting in considerable cost savings. For example, with little effort, it is possible to migrate from Windows to Linux to a bare machine without requiring any changes to the code. Thread management and synchronization is all done by the implementation, transparently. However, in some situations, it's critical to be able to access directly the services provided by the platform. In this case, it's always possible to make direct system calls from Ada code. Several targets of the GNAT compiler provide this sort of API by default, for example win32ada for Windows and Florist for POSIX systems.

On native and RTOS-based platforms GNAT typically provides the full concurrency facilities. In contrast, on bare metal platforms GNAT typically provides the two standard subsets: Ravenscar and Jorvik.

64.2 Tasks

Ada offers a high level construct called a *task* which is an independent thread of execution. In GNAT, tasks are either mapped to the underlying OS threads, or use a dedicated kernel when not available.

The following example will display the 26 letters of the alphabet twice, using two concurrent tasks. Since there is no synchronization between the two threads of control in any of the examples, the output may be interspersed.

[Ada]

Listing 1: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is -- implicitly called by the environment task
4      subtype A_To_Z is Character range 'A' .. 'Z';
5
6      task My_Task;
7
8          task body My_Task is
9              begin
10                 for I in A_To_Z'Range loop
11                     Put (I);
12                 end loop;
13                 New_Line;
14             end My_Task;
15         begin
16             for I in A_To_Z'Range loop
17                 Put (I);
18                 end loop;
19                 New_Line;
20         end Main;

```

Runtime output

```

ABCDEFGHIJKLMNOPQRSTUVWXYZ
MNOPQRSTUVWXYZ

```

Any number of Ada tasks may be declared in any declarative region. A task declaration is very similar to a procedure or package declaration. They all start automatically when control reaches the begin. A block will not exit until all sequences of statements defined within that scope, including those in tasks, have been completed.

A task type is a generalization of a task object; each object of a task type has the same behavior. A declared object of a task type is started within the scope where it is declared, and control does not leave that scope until the task has terminated.

Task types can be parameterized; the parameter serves the same purpose as an argument to a constructor in Java. The following example creates 10 tasks, each of which displays a subset of the alphabet contained between the parameter and the 'Z' Character. As with the earlier example, since there is no synchronization among the tasks, the output may be interspersed depending on the underlying implementation of the task scheduling algorithm.

[Ada]

Listing 2: my_tasks.ads

```

1  package My_Tasks is
2
3      task type My_Task (First : Character);
4
5  end My_Tasks;

```

Listing 3: my_tasks.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body My_Tasks is
4
5      task body My_Task is
6          begin

```

(continues on next page)

(continued from previous page)

```

7   for I in First .. 'Z' loop
8     Put (I);
9   end loop;
10  New_Line;
11  end My_Task;
12
13 end My_Tasks;

```

Listing 4: main.adb

```

1 with My_Tasks; use My_Tasks;
2
3 procedure Main is
4   Dummy_Tab : array (0 .. 3) of My_Task ('W');
5 begin
6   null;
7 end Main;

```

Runtime output

```

WXYZ
WXYZ
WXYZ
WXYZ

```

In Ada, a task may be dynamically allocated rather than declared statically. The task will then start as soon as it has been allocated, and terminates when its work is completed.

[Ada]

Listing 5: main.adb

```

1 with My_Tasks; use My_Tasks;
2
3 procedure Main is
4   type Ptr_Task is access My_Task;
5
6   T : Ptr_Task;
7 begin
8   T := new My_Task ('W');
9 end Main;

```

Build output

```

main.adb:6:04: warning: variable "T" is assigned but never read [-gnatwm]
main.adb:8:04: warning: possibly useless assignment to "T", value might not be
  referenced [-gnatwm]

```

Runtime output

```

WXYZ

```

64.3 Rendezvous

A rendezvous is a synchronization between two tasks, allowing them to exchange data and coordinate execution. Let's consider the following example:

[Ada]

Listing 6: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      task After is
6          entry Go;
7      end After;
8
9      task body After is
10     begin
11         accept Go;
12         Put_Line ("After");
13     end After;
14
15 begin
16     Put_Line ("Before");
17     After.Go;
18 end Main;

```

Runtime output

```
Before
After
```

The Go entry declared in After is the client interface to the task. In the task body, the **accept** statement causes the task to wait for a call on the entry. This particular **entry** and **accept** pair simply causes the task to wait until Main calls After.Go. So, even though the two tasks start simultaneously and execute independently, they can coordinate via Go. Then, they both continue execution independently after the rendezvous.

The **entry/accept** pair can take/pass parameters, and the **accept** statement can contain a sequence of statements; while these statements are executed, the caller is blocked.

Let's look at a more ambitious example. The rendezvous below accepts parameters and executes some code:

[Ada]

Listing 7: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      task After is
6          entry Go (Text : String);
7      end After;
8
9      task body After is
10     begin
11         accept Go (Text : String) do
12             Put_Line ("After: " & Text);
13     end Go;

```

(continues on next page)

(continued from previous page)

```

14    end After;
15
16 begin
17   Put_Line ("Before");
18   After.Go ("Main");
19 end Main;

```

Runtime output

```

Before
After: Main

```

In the above example, the Put_Line is placed in the accept statement. Here's a possible execution trace, assuming a uniprocessor:

1. At the begin of Main, task After is started and the main procedure is suspended.
2. After reaches the **accept** statement and is suspended, since there is no pending call on the Go entry.
3. The main procedure is awakened and executes the Put_Line invocation, displaying the string "Before".
4. The main procedure calls the Go entry. Since After is suspended on its **accept** statement for this entry, the call succeeds.
5. The main procedure is suspended, and the task After is awakened to execute the body of the **accept** statement. The actual parameter "Main" is passed to the **accept** statement, and the Put_Line invocation is executed. As a result, the string "After: Main" is displayed.
6. When the **accept** statement is completed, both the After task and the main procedure are ready to run. Suppose that the Main procedure is given the processor. It reaches its end, but the local task After has not yet terminated. The main procedure is suspended.
7. The After task continues, and terminates since it is at its end. The main procedure is resumed, and it too can terminate since its dependent task has terminated.

The above description is a conceptual model; in practice the implementation can perform various optimizations to avoid unnecessary context switches.

64.4 Selective Rendezvous

The **accept** statement by itself can only wait for a single event (call) at a time. The **select** statement allows a task to listen for multiple events simultaneously, and then to deal with the first event to occur. This feature is illustrated by the task below, which maintains an integer value that is modified by other tasks that call Increment, Decrement, and Get:

[Ada]

Listing 8: counters.ads

```

1 package Counters is
2
3   task Counter is
4     entry Get (Result : out Integer);
5     entry Increment;
6     entry Decrement;
7   end Counter;

```

(continues on next page)

(continued from previous page)

```
8 end Counters;
```

Listing 9: counters.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Counters is
4
5   task body Counter is
6     Value : Integer := 0;
7   begin
8     loop
9       select
10         accept Increment do
11           Value := Value + 1;
12         end Increment;
13       or
14         accept Decrement do
15           Value := Value - 1;
16         end Decrement;
17       or
18         accept Get (Result : out Integer) do
19           Result := Value;
20         end Get;
21       or
22         delay 5.0;
23         Put_Line ("Exiting Counter task...");
24         exit;
25       end select;
26     end loop;
27   end Counter;
28
29 end Counters;
```

Listing 10: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Counters;    use Counters;
3
4 procedure Main is
5   V : Integer;
6 begin
7   Put_Line ("Main started.");
8
9   Counter.Get (V);
10  Put_Line ("Got value. Value = " & Integer'Image (V));
11
12  Counter.Increment;
13  Put_Line ("Incremented value.");
14
15  Counter.Increment;
16  Put_Line ("Incremented value.");
17
18  Counter.Get (V);
19  Put_Line ("Got value. Value = " & Integer'Image (V));
20
21  Counter.Decrement;
22  Put_Line ("Decrement value.");
```

(continues on next page)

(continued from previous page)

```

24 Counter.Get (V);
25 Put_Line ("Got value. Value = " & Integer'Image (V));
26
27 Put_Line ("Main finished.");
28 end Main;

```

Runtime output

```

Main started.
Got value. Value = 0
Incremented value.
Incremented value.
Got value. Value = 2
Decrement value.
Got value. Value = 1
Main finished.
Exiting Counter task...

```

When the task's statement flow reaches the select, it will wait for all four events — three entries and a delay — in parallel. If the delay of five seconds is exceeded, the task will execute the statements following the **delay** statement (and in this case will exit the loop, in effect terminating the task). The **accept** bodies for the Increment, Decrement, or Get entries will be otherwise executed as they're called. These four sections of the select statement are mutually exclusive: at each iteration of the loop, only one will be invoked. This is a critical point; if the task had been written as a package, with procedures for the various operations, then a *race condition* could occur where multiple tasks simultaneously calling, say, Increment, cause the value to only get incremented once. In the tasking version, if multiple tasks simultaneously call Increment then only one at a time will be accepted, and the value will be incremented by each of the tasks when it is accepted.

More specifically, each entry has an associated queue of pending callers. If a task calls one of the entries and Counter is not ready to accept the call (i.e., if Counter is not suspended at the **select** statement) then the calling task is suspended, and placed in the queue of the entry that it is calling. From the perspective of the Counter task, at any iteration of the loop there are several possibilities:

- There is no call pending on any of the entries. In this case Counter is suspended. It will be awakened by the first of two events: a call on one of its entries (which will then be immediately accepted), or the expiration of the five second delay (whose effect was noted above).
- There is a call pending on exactly one of the entries. In this case control passes to the **select** branch with an **accept** statement for that entry.
- There are calls pending on more than one entry. In this case one of the entries with pending callers is chosen, and then one of the callers is chosen to be de-queued. The choice of which caller to accept depends on the queuing policy, which can be specified via a **pragma** defined in the Real-Time Systems Annex of the Ada standard; the default is *First-In First-Out*.

64.5 Protected Objects

Although the rendezvous may be used to implement mutually exclusive access to a shared data object, an alternative (and generally preferable) style is through a protected object, an efficiently implementable mechanism that makes the effect more explicit. A protected object has a public interface (its protected operations) for accessing and manipulating the object's components (its private part). Mutual exclusion is enforced through a conceptual lock on the object, and encapsulation ensures that the only external access to the components are through the protected operations.

Two kinds of operations can be performed on such objects: read-write operations by procedures or entries, and read-only operations by functions. The lock mechanism is implemented so that it's possible to perform concurrent read operations but not concurrent write or read/write operations.

Let's reimplement our earlier tasking example with a protected object called Counter:

[Ada]

Listing 11: counters.ads

```

1 package Counters is
2
3     protected Counter is
4         function Get return Integer;
5         procedure Increment;
6         procedure Decrement;
7     private
8         Value : Integer := 0;
9     end Counter;
10
11 end Counters;
```

Listing 12: counters.adb

```

1 package body Counters is
2
3     protected body Counter is
4         function Get return Integer is
5             begin
6                 return Value;
7             end Get;
8
9         procedure Increment is
10            begin
11                Value := Value + 1;
12            end Increment;
13
14         procedure Decrement is
15            begin
16                Value := Value - 1;
17            end Decrement;
18     end Counter;
19
20 end Counters;
```

Having two completely different ways to implement the same paradigm might seem complicated. However, in practice the actual problem to solve usually drives the choice between an active structure (a task) or a passive structure (a protected object).

A protected object can be accessed through prefix notation:

[Ada]

Listing 13: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2  with Counters;    use Counters;
3
4  procedure Main is
5  begin
6      Counter.Increment;
7      Counter.Decrement;
8      Put_Line (Integer'Image (Counter.Get));
9  end Main;

```

Runtime output

```
0
```

A protected object may look like a package syntactically, since it contains declarations that can be accessed externally using prefix notation. However, the declaration of a protected object is extremely restricted; for example, no public data is allowed, no types can be declared inside, etc. And besides the syntactic differences, there is a critical semantic distinction: a protected object has a conceptual lock that guarantees mutual exclusion; there is no such lock for a package.

Like tasks, it's possible to declare protected types that can be instantiated several times:

```

declare
  protected type Counter is
    -- as above
  end Counter;

  protected body Counter is
    -- as above
  end Counter;

  C1 : Counter;
  C2 : Counter;
begin
  C1.Increment;
  C2.Decrement;
  . . .
end;

```

Protected objects and types can declare a procedure-like operation known as an *entry*. An entry is somewhat similar to a procedure but includes a so-called barrier condition that must be true in order for the entry invocation to succeed. Calling a protected entry is thus a two step process: first, acquire the lock on the object, and then evaluate the barrier condition. If the condition is true then the caller will execute the entry body. If the condition is false, then the caller is placed in the queue for the entry, and relinquishes the lock. Barrier conditions (for entries with non-empty queues) are reevaluated upon completion of protected procedures and protected entries.

Here's an example illustrating protected entries: a protected type that models a binary semaphore / persistent signal.

[Ada]

Listing 14: binary_semaphores.ads

```

1  package Binary_Semaphores is
2

```

(continues on next page)

(continued from previous page)

```

3   protected type Binary_Semaphore is
4     entry Wait;
5     procedure Signal;
6   private
7     Signaled : Boolean := False;
8   end Binary_Semaphore;
9
10 end Binary_Semaphores;

```

Listing 15: binary_semaphores.adb

```

1  package body Binary_Semaphores is
2
3    protected body Binary_Semaphore is
4      entry Wait when Signaled is
5        begin
6          Signaled := False;
7        end Wait;
8
9        procedure Signal is
10       begin
11         Signaled := True;
12       end Signal;
13     end Binary_Semaphore;
14
15 end Binary_Semaphores;

```

Listing 16: main.adb

```

1  with Ada.Text_Io;           use Ada.Text_Io;
2  with Binary_Semaphores; use Binary_Semaphores;
3
4  procedure Main is
5    B : Binary_Semaphore;
6
7    task T1;
8    task T2;
9
10   task body T1 is
11     begin
12       Put_Line ("Task T1 waiting...");
13       B.Wait;
14
15       Put_Line ("Task T1.");
16       delay 1.0;
17
18       Put_Line ("Task T1 will signal...");
19       B.Signal;
20
21       Put_Line ("Task T1 finished.");
22     end T1;
23
24   task body T2 is
25     begin
26       Put_Line ("Task T2 waiting...");
27       B.Wait;
28
29       Put_Line ("Task T2");
30       delay 1.0;
31

```

(continues on next page)

(continued from previous page)

```

32     Put_Line ("Task T2 will signal...");
33     B.Signal;
34
35     Put_Line ("Task T2 finished.");
36   end T2;
37
38 begin
39   Put_Line ("Main started.");
40   B.Signal;
41   Put_Line ("Main finished.");
42 end Main;

```

Runtime output

```

Task T1 waiting...
Task T2 waiting...
Main started.
Main finished.
Task T1.
Task T1 will signal...
Task T1 finished.
Task T2
Task T2 will signal...
Task T2 finished.

```

Ada concurrency features provide much further generality than what's been presented here. For additional information please consult one of the works cited in the *References* section.

64.6 Ravenscar

The Ravenscar profile is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are intended:

- hard real-time applications requiring predictability,
- safety-critical systems requiring formal, stringent certification,
- high-integrity applications requiring formal static analysis and verification,
- embedded applications requiring both a small memory footprint and low execution overhead.

Tasking constructs that preclude analysis, either technically or economically, are disallowed. You can use the **pragma Profile** (Ravenscar) to indicate that the Ravenscar restrictions must be observed in your program.

Some of the examples we've seen above will be rejected by the compiler when using the Ravenscar profile. For example:

[Ada]

Listing 17: my_tasks.ads

```

1 package My_Tasks is
2
3   task type My_Task (First : Character);
4
5 end My_Tasks;

```

Listing 18: my_tasks.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 package body My_Tasks is
4
5   task body My_Task is
6     begin
7       for C in First .. 'Z' loop
8         Put (C);
9       end loop;
10      New_Line;
11    end My_Task;
12
13 end My_Tasks;
```

Listing 19: main.adb

```
1 pragma Profile (Ravenscar);
2
3 with My_Tasks; use My_Tasks;
4
5 procedure Main is
6   Tab : array (0 .. 3) of My_Task ('W');
7 begin
8   null;
9 end Main;
```

Build output

```
main.adb:6:04: error: violation of restriction "No_Task_Hierarchy"
main.adb:6:04: error: from profile "Ravenscar" at line 1
gprbuild: *** compilation phase failed
```

This code violates the *No_Task_Hierarchy* restriction of the Ravenscar profile. This is due to the declaration of Tab in the Main procedure. Ravenscar requires task declarations to be done at the library level. Therefore, a simple solution is to create a separate package and reference it in the main application:

[Ada]

Listing 20: my_task_inst.ads

```
1 with My_Tasks; use My_Tasks;
2
3 package My_Task_Inst is
4
5   Tab : array (0 .. 3) of My_Task ('W');
6
7 end My_Task_Inst;
```

Listing 21: main.adb

```
1 pragma Profile (Ravenscar);
2
3 with My_Task_Inst;
4
5 procedure Main is
6 begin
7   null;
8 end Main;
```

Build output

```
main.adb:3:06: warning: unit "My_Task_Inst" is not referenced [-gnatwu]
```

Runtime output

```
WXYZ  
WXYZ  
WXYZ  
WXYZ
```

Also, Ravenscar prohibits entries for tasks. For example, we're not allowed to write this declaration:

```
task type My_Task (First : Character) is
  entry Start;
end My_Task;
```

You can use, however, one entry per protected object. As an example, the declaration of the Binary_Semaphore type that we've discussed before compiles fine with Ravenscar:

```
protected type Binary_Semaphore is
  entry Wait;
  procedure Signal;
private
  Signaled : Boolean := False;
end Binary_Semaphore;
```

We could add more procedures and functions to the declaration of Binary_Semaphore, but we wouldn't be able to add another entry when using Ravenscar.

Similar to the previous example with the task array declaration, objects of Binary_Semaphore cannot be declared in the main application:

```
procedure Main is
  B : Binary_Semaphore;
begin
  null;
end Main;
```

This violates the *No_Local_Protected_Objects* restriction. Again, Ravenscar expects this declaration to be done on a library level, so a solution to make this code compile is to have this declaration in a separate package and reference it in the Main procedure.

Ravenscar offers many additional restrictions. Covering those would exceed the scope of this chapter. You can find more examples using the Ravenscar profile on [this blog post¹⁰⁰](https://blog.adacore.com/theres-a-mini-rtos-in-my-language).

¹⁰⁰ <https://blog.adacore.com/theres-a-mini-rtos-in-my-language>

WRITING ADA ON EMBEDDED SYSTEMS

65.1 Understanding the Ada Run-Time

Ada supports a high level of abstractness and expressiveness. In some cases, the compiler translates those constructs directly into machine code. However, there are many high-level constructs for which a direct compilation would be difficult. In those cases, the compiler links to a library containing an implementation of those high-level constructs: this is the so-called run-time library.

One typical example of high-level constructs that can be cumbersome for direct machine code generation is Ada source-code using tasking. In this case, linking to a low-level implementation of multithreading support — for example, an implementation using POSIX threads — is more straightforward than trying to make the compiler generate all the machine code.

In the case of GNAT, the run-time library is implemented using both C and Ada source-code. Also, depending on the operating system, the library will interface with low-level functionality from the target operating system.

There are basically two types of run-time libraries:

- the **standard** run-time library: in many cases, this is the run-time library available on desktop operating systems or on some embedded platforms (such as ARM-Linux on a Raspberry-Pi).
- the **configurable** run-time library: this is a capability that is used to create custom run-time libraries for specific target devices.

Configurable run-time libraries are usually used for constrained target devices where support for the full library would be difficult or even impossible. In this case, configurable run-time libraries may support just a subset of the full Ada language. There are many reasons that speak for this approach:

- Some aspects of the Ada language may not translate well to limited operating systems.
- Memory constraints may require reducing the size of the run-time library, so that developers may need to replace or even remove parts of the library.
- When certification is required, those parts of the library that would require too much certification effort can be removed.

When using a configurable run-time library, the compiler checks whether the library supports certain features of the language. If a feature isn't supported, the compiler will give an error message.

You can find further information about the run-time library on [this chapter of the GNAT User's Guide Supplement for Cross Platforms](#)¹⁰¹

¹⁰¹ https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx/the_gnat_configurable_run_time_facility.html

65.2 Low Level Programming

65.2.1 Representation Clauses

We've seen in the previous chapters how Ada can be used to describe high level semantics and architecture. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded assembly code or bit-level data management.

One very interesting feature of the language is that, unlike C, for example, there are no data representation constraints unless specified by the developer. This means that the compiler is free to choose the best trade-off in terms of representation vs. performance. Let's start with the following example:

[Ada]

```
type R is record
  V : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record
with Pack;
```

[C]

```
struct R {
  unsigned int v:8;
  bool b1;
  bool b2;
};
```

The Ada and the C code above both represent efforts to create an object that's as small as possible. Controlling data size is not possible in Java, but the language does specify the size of values for the primitive types.

Although the C and Ada code are equivalent in this particular example, there's an interesting semantic difference. In C, the number of bits required by each field needs to be specified. Here, we're stating that v is only 8 bits, effectively representing values from 0 to 255. In Ada, it's the other way around: the developer specifies the range of values required and the compiler decides how to represent things, optimizing for speed or size. The Pack aspect declared at the end of the record specifies that the compiler should optimize for size even at the expense of decreased speed in accessing record components. We'll see more details about the Pack aspect in the sections about [bitwise operations](#) (page 752) and [mapping structures to bit-fields](#) (page 754) in chapter 6.

Other representation clauses can be specified as well, along with compile-time consistency checks between requirements in terms of available values and specified sizes. This is particularly useful when a specific layout is necessary; for example when interfacing with hardware, a driver, or a communication protocol. Here's how to specify a specific data layout based on the previous example:

[Ada]

```
type R is record
  V : Integer range 0 .. 255;
  B1 : Boolean;
  B2 : Boolean;
end record;

for R use record
  -- Occupy the first bit of the first byte.
```

(continues on next page)

(continued from previous page)

```

B1 at 0 range 0 .. 0;
  -- Occupy the last 7 bits of the first byte,
  -- as well as the first bit of the second byte.
V at 0 range 1 .. 8;
  -- Occupy the second bit of the second byte.
B2 at 1 range 1 .. 1;
end record;

```

We omit the `with` Pack directive and instead use a record representation clause following the record declaration. The compiler is directed to spread objects of type R across two bytes. The layout we're specifying here is fairly inefficient to work with on any machine, but you can have the compiler construct the most efficient methods for access, rather than coding your own machine-dependent bit-level methods manually.

65.2.2 Embedded Assembly Code

When performing low-level development, such as at the kernel or hardware driver level, there can be times when it is necessary to implement functionality with assembly code.

Every Ada compiler has its own conventions for embedding assembly code, based on the hardware platform and the supported assembler(s). Our examples here will work with GNAT and GCC on the x86 architecture.

All x86 processors since the Intel Pentium offer the `rdtsc` instruction, which tells us the number of cycles since the last processor reset. It takes no inputs and places an unsigned 64-bit value split between the `edx` and `eax` registers.

GNAT provides a subprogram called `System.Machine_Code.Asm` that can be used for assembly code insertion. You can specify a string to pass to the assembler as well as source-level variables to be used for input and output:

[Ada]

Listing 1: `get_processor_cycles.adb`

```

1  with System.Machine_Code; use System.Machine_Code;
2  with Interfaces;           use Interfaces;
3
4  function Get_Processor_Cycles return Unsigned_64 is
5    Low, High : Unsigned_32;
6    Counter   : Unsigned_64;
7  begin
8    Asm ("rdtsc",
9          Outputs =>
10            (Unsigned_32'Asm_Output ("=a", High),
11             Unsigned_32'Asm_Output ("=d", Low)),
12             Volatile => True);
13
14    Counter :=
15      Unsigned_64 (High) * 2 ** 32 +
16      Unsigned_64 (Low);
17
18    return Counter;
19  end Get_Processor_Cycles;

```

The `Unsigned_32'Asm_Output` clauses above provide associations between machine registers and source-level variables to be updated. `=a` and `=d` refer to the `eax` and `edx` machine registers, respectively. The use of the `Unsigned_32` and `Unsigned_64` types from package

Interfaces ensures correct representation of the data. We assemble the two 32-bit values to form a single 64-bit value.

We set the Volatile parameter to **True** to tell the compiler that invoking this instruction multiple times with the same inputs can result in different outputs. This eliminates the possibility that the compiler will optimize multiple invocations into a single call.

With optimization turned on, the GNAT compiler is smart enough to use the eax and edx registers to implement the High and Low variables, resulting in zero overhead for the assembly interface.

The machine code insertion interface provides many features beyond what was shown here. More information can be found in the GNAT User's Guide, and the GNAT Reference manual.

65.3 Interrupt Handling

Handling interrupts is an important aspect when programming embedded devices. Interrupts are used, for example, to indicate that a hardware or software event has happened. Therefore, by handling interrupts, an application can react to external events.

Ada provides built-in support for handling interrupts. We can process interrupts by attaching a handler — which must be a protected procedure — to it. In the declaration of the protected procedure, we use the `Attach_Handler` aspect and indicate which interrupt we want to handle.

Let's look into a code example that *traps* the quit interrupt (SIGQUIT) on Linux:

[Ada]

Listing 2: signal_handlers.ads

```
1  with System.OS_Interface;
2
3  package Signal_Handlers is
4
5      protected type Quit_Handler is
6          function Requested return Boolean;
7      private
8          Quit_Request : Boolean := False;
9
10     -- 
11     -- Declaration of an interrupt handler for the "quit" interrupt:
12     --
13     procedure Handle_Quit_Signal
14         with Attach_Handler => System.OS_Interface.SIGQUIT;
15 end Quit_Handler;
16
17 end Signal_Handlers;
```

Listing 3: signal_handlers.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Signal_Handlers is
4
5      protected body Quit_Handler is
6
7          function Requested return Boolean is
8              (Quit_Request);
9
```

(continues on next page)

(continued from previous page)

```

10  procedure Handle_Quit_Signal is
11    begin
12      Put_Line ("Quit request detected!");
13      Quit_Request := True;
14      end Handle_Quit_Signal;
15
16  end Quit_Handler;
17
18 end Signal_Handlers;

```

Listing 4: test_quit_handler.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2  with Signal_Handlers;
3
4  procedure Test_Quit_Handler is
5    Quit : Signal_Handlers.Quit_Handler;
6
7  begin
8    while True loop
9      delay 1.0;
10     exit when Quit.Requested;
11   end loop;
12
13   Put_Line ("Exiting application...");
14 end Test_Quit_Handler;

```

The specification of the `Signal_Handlers` package from this example contains the declaration of `Quit_Handler`, which is a protected type. In the private part of this protected type, we declare the `Handle_Quit_Signal` procedure. By using the `Attach_Handler` aspect in the declaration of `Handle_Quit_Signal` and indicating the quit interrupt (`System.OS_Interface.SIGQUIT`), we're instructing the operating system to call this procedure for any quit request. So when the user presses `CTRL+\` on their keyboard, for example, the application will behave as follows:

- the operating system calls the `Handle_Quit_Signal` procedure , which displays a message to the user ("Quit request detected!") and sets a Boolean variable — `Quit_Request`, which is declared in the `Quit_Handler` type;
- the main application checks the status of the quit handler by calling the `Requested` function as part of the `while True` loop;
 - This call is in the `exit when Quit.Requested` line.
 - The `Requested` function returns `True` in this case because the `Quit_Request` flag was set by the `Handle_Quit_Signal` procedure.
- the main applications exits the loop, displays a message and finishes.

Note that the code example above isn't portable because it makes use of interrupts from the Linux operating system. When programming embedded devices, we would use instead the interrupts available on those specific devices.

Also note that, in the example above, we're declaring a static handler at compilation time. If you need to make use of dynamic handlers, which can be configured at runtime, you can use the subprograms from the `Ada.Interrupts` package. This package includes not only a version of `Attach_Handler` as a procedure, but also other procedures such as:

- `Exchange_Handler`, which lets us exchange, at runtime, the current handler associated with a specific interrupt by a different handler;
- `Detach_Handler`, which we can use to remove the handler currently associated with a given interrupt.

Details about the Ada.Interrupts package are out of scope for this course. We'll discuss them in a separate, more advanced course in the future. You can find some information about it in the [Interrupts appendix of the Ada Reference Manual](#)¹⁰².

65.4 Dealing with Absence of FPU with Fixed Point

Many numerical applications typically use floating-point types to compute values. However, in some platforms, a floating-point unit may not be available. Other platforms may have a floating-point unit, but using it in certain numerical algorithms can be prohibitive in terms of performance. For those cases, fixed-point arithmetic can be a good alternative.

The difference between fixed-point and floating-point types might not be so obvious when looking at this code snippet:

[Ada]

Listing 5: fixed_definitions.ads

```
1 package Fixed_Definitions is
2
3     D : constant := 2.0 ** (-31);
4
5     type Fixed is delta D range -1.0 .. 1.0 - D;
6
7 end Fixed_Definitions;
```

Listing 6: show_float_and_fixed_point.adb

```
1 with Ada.Text_Io;           use Ada.Text_Io;
2
3 with Fixed_Definitions;   use Fixed_Definitions;
4
5 procedure Show_Float_And_Fixed_Point is
6     Float_Value : Float := 0.25;
7     Fixed_Value : Fixed := 0.25;
8 begin
9
10    Float_Value := Float_Value + 0.25;
11    Fixed_Value := Fixed_Value + 0.25;
12
13    Put_Line ("Float_Value = " & Float'Image (Float_Value));
14    Put_Line ("Fixed_Value = " & Fixed'Image (Fixed_Value));
15 end Show_Float_And_Fixed_Point;
```

Runtime output

```
Float_Value = 5.00000E-01
Fixed_Value = 0.5000000000
```

In this example, the application will show the value 0.5 for both `Float_Value` and `Fixed_Value`.

The major difference between floating-point and fixed-point types is in the way the values are stored. Values of ordinary fixed-point types are, in effect, scaled integers. The scaling used for ordinary fixed-point types is defined by the type's `delta`, which is derived from the specified `delta` and, by default, is a power of two. Therefore, ordinary fixed-point types are sometimes called binary fixed-point types. In that sense, ordinary fixed-point types can

¹⁰² <http://www.adc-auth.org/standards/12aarm/html/AA-C-3-2.html>

be thought of being close to the actual representation on the machine. In fact, ordinary fixed-point types make use of the available integer shift instructions, for example.

Another difference between floating-point and fixed-point types is that Ada doesn't provide standard fixed-point types — except for the **Duration** type, which is used to represent an interval of time in seconds. While the Ada standard specifies floating-point types such as **Float** and **Long_Float**, we have to declare our own fixed-point types. Note that, in the previous example, we have used a fixed-point type named **Fixed**: this type isn't part of the standard, but must be declared somewhere in the source-code of our application.

The syntax for an ordinary fixed-point type is

```
type <type_name> is delta <delta_value> range <lower_bound> .. <upper_bound>;
```

By default, the compiler will choose a scale factor, or **small**, that is a power of 2 no greater than <delta_value>.

For example, we may define a normalized range between -1.0 and 1.0 as following:

[Ada]

Listing 7: normalized_fixed_point_type.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Normalized_Fixed_Point_Type is
4   D : constant := 2.0 ** (-31);
5   type TQ31 is delta D range -1.0 .. 1.0 - D;
6 begin
7   Put_Line ("TQ31 requires " & Integer'Image (TQ31'Size) & " bits");
8   Put_Line ("The delta value of TQ31 is " & TQ31'Image (TQ31'Delta));
9   Put_Line ("The minimum value of TQ31 is " & TQ31'Image (TQ31'First));
10  Put_Line ("The maximum value of TQ31 is " & TQ31'Image (TQ31'Last));
11 end Normalized_Fixed_Point_Type;
```

Runtime output

```
TQ31 requires 32 bits
The delta value of TQ31 is 0.0000000005
The minimum value of TQ31 is -1.0000000000
The maximum value of TQ31 is 0.9999999995
```

In this example, we are defining a 32-bit fixed-point data type for our normalized range. When running the application, we notice that the upper bound is close to one, but not exactly one. This is a typical effect of fixed-point data types — you can find more details in this discussion about the **Q format**¹⁰³. We may also rewrite this code with an exact type definition:

[Ada]

Listing 8: normalized_adapted_fixed_point_type.ads

```
1 package Normalized_Adapted_Fixed_Point_Type is
2
3   type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
4
5 end Normalized_Adapted_Fixed_Point_Type;
```

We may also use any other range. For example:

[Ada]

¹⁰³ [https://en.wikipedia.org/wiki/Q_\(number_format\)](https://en.wikipedia.org/wiki/Q_(number_format))

Listing 9: custom_fixed_point_range.adb

```

1  with Ada.Text_Io;  use Ada.Text_Io;
2  with Ada.Numerics; use Ada.Numerics;
3
4  procedure Custom_Fixed_Point_Range is
5    type Inv_Trig is delta 2.0 ** (-15) * Pi range -Pi / 2.0 .. Pi / 2.0;
6  begin
7    Put_Line ("Inv_Trig requires " & Integer'Image (Inv_Trig'Size)
8              & " bits");
9    Put_Line ("The delta value of Inv_Trig is "
10              & Inv_Trig'Image (Inv_Trig'Delta));
11   Put_Line ("The minimum value of Inv_Trig is "
12              & Inv_Trig'Image (Inv_Trig'First));
13   Put_Line ("The maximum value of Inv_Trig is "
14              & Inv_Trig'Image (Inv_Trig'Last));
15 end Custom_Fixed_Point_Range;

```

Build output

```
custom_fixed_point_range.adb:10:40: warning: static fixed-point value is not a
multiple of Small [-gnatwb]
```

Runtime output

```

Inv_Trig requires 16 bits
The delta value of Inv_Trig is 0.000006
The minimum value of Inv_Trig is -1.57080
The maximum value of Inv_Trig is 1.57080

```

In this example, we are defining a 16-bit type called Inv_Trig, which has a range from $-\pi/2$ to $\pi/2$.

All standard operations are available for fixed-point types. For example:

[Ada]

Listing 10: fixed_point_op.adb

```

1  with Ada.Text_Io;  use Ada.Text_Io;
2
3  procedure Fixed_Point_Op is
4    type TQ31 is delta 2.0 ** (-31) range -1.0 .. 1.0 - 2.0 ** (-31);
5
6    A, B, R : TQ31;
7  begin
8    A := 0.25;
9    B := 0.50;
10   R := A + B;
11   Put_Line ("R is " & TQ31'Image (R));
12 end Fixed_Point_Op;

```

Runtime output

```
R is 0.75000000000
```

As expected, R contains 0.75 after the addition of A and B.

In the case of C, since the language doesn't support fixed-point arithmetic, we need to emulate it using integer types and custom operations via functions. Let's look at this very rudimentary example:

[C]

Listing 11: main.c

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define SHIFT_FACTOR 32
5
6 #define T0_FIXED(x) ((int) ((x) * pow (2.0, SHIFT_FACTOR - 1)))
7 #define T0_FLOAT(x) ((float) ((double)(x) * (double)pow (2.0, -(SHIFT_FACTOR - 1))))
8
9 typedef int fixed;
10
11 fixed add (fixed a, fixed b)
12 {
13     return a + b;
14 }
15
16 fixed mult (fixed a, fixed b)
17 {
18     return (fixed)((long)a * (long)b) >> (SHIFT_FACTOR - 1);
19 }
20
21 void display_fixed (fixed x)
22 {
23     printf("value (integer) = %d\n", x);
24     printf("value (float)   = %3.5f\n\n", T0_FLOAT (x));
25 }
26
27 int main(int argc, const char * argv[])
28 {
29     int fixed_value = T0_FIXED(0.25);
30
31     printf("Original value\n");
32     display_fixed(fixed_value);
33
34     printf("... + 0.25\n");
35     fixed_value = add(fixed_value, T0_FIXED(0.25));
36     display_fixed(fixed_value);
37
38     printf("... * 0.5\n");
39     fixed_value = mult(fixed_value, T0_FIXED(0.5));
40     display_fixed(fixed_value);
41
42     return 0;
43 }
```

Runtime output

```

Original value
value (integer) = 536870912
value (float)   = 0.25000

... + 0.25
value (integer) = 1073741824
value (float)   = 0.50000

... * 0.5
value (integer) = 536870912
value (float)   = 0.25000
```

Here, we declare the fixed-point type `fixed` based on `int` and two operations for it: addition (via the `add` function) and multiplication (via the `mult` function). Note that, while fixed-point addition is quite straightforward, multiplication requires right-shifting to match the correct internal representation. In Ada, since fixed-point operations are part of the language specification, they don't need to be emulated. Therefore, no extra effort is required from the programmer.

Also note that the example above is very rudimentary, so it doesn't take some of the side-effects of fixed-point arithmetic into account. In C, you have to manually take all side-effects deriving from fixed-point arithmetic into account, while in Ada, the compiler takes care of selecting the right operations for you.

65.5 Volatile and Atomic data

Ada has built-in support for handling both volatile and atomic data. Let's start by discussing volatile objects.

65.5.1 Volatile

A `volatile`¹⁰⁴ object can be described as an object in memory whose value may change between two consecutive memory accesses of a process A — even if process A itself hasn't changed the value. This situation may arise when an object in memory is being shared by multiple threads. For example, a thread *B* may modify the value of that object between two read accesses of a thread *A*. Another typical example is the one of `memory-mapped I/O`¹⁰⁵, where the hardware might be constantly changing the value of an object in memory.

Because the value of a volatile object may be constantly changing, a compiler cannot generate code that stores the value of that object into a register and use the value from the register in subsequent operations. Storing into a register is avoided because, if the value is stored there, it would be outdated if another process had changed the volatile object in the meantime. Instead, the compiler generates code in such a way that the process must read the value of the volatile object from memory for each access.

Let's look at a simple example of a volatile variable in C:

[C]

Listing 12: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     volatile double val = 0.0;
6     int i;
7
8     for (i = 0; i < 1000; i++)
9     {
10         val += i * 2.0;
11     }
12     printf ("val: %5.3f\n", val);
13
14     return 0;
15 }
```

Runtime output

¹⁰⁴ [https://en.wikipedia.org/wiki/Volatile_\(computer_programming\)](https://en.wikipedia.org/wiki/Volatile_(computer_programming))

¹⁰⁵ https://en.wikipedia.org/wiki/Memory-mapped_I/O

```
val: 999000.000
```

In this example, val has the modifier **volatile**, which indicates that the compiler must handle val as a volatile object. Therefore, each read and write access in the loop is performed by accessing the value of val in then memory.

This is the corresponding implementation in Ada:

[Ada]

Listing 13: show_volatile_object.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Volatile_Object is
4      Val : Long_Float with Volatile;
5  begin
6      Val := 0.0;
7      for I in 0 .. 999 loop
8          Val := Val + 2.0 * Long_Float (I);
9      end loop;
10
11     Put_Line ("Val: " & Long_Float'Image (Val));
12 end Show_Volatile_Object;
```

Runtime output

```
Val: 9.99000000000000E+05
```

In this example, Val has the Volatile aspect, which makes the object volatile. We can also use the Volatile aspect in type declarations. For example:

[Ada]

Listing 14: show_volatile_type.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Volatile_Type is
4      type Volatile_Long_Float is new Long_Float with Volatile;
5
6      Val : Volatile_Long_Float;
7  begin
8      Val := 0.0;
9      for I in 0 .. 999 loop
10         Val := Val + 2.0 * Volatile_Long_Float (I);
11     end loop;
12
13     Put_Line ("Val: " & Volatile_Long_Float'Image (Val));
14 end Show_Volatile_Type;
```

Runtime output

```
Val: 9.99000000000000E+05
```

Here, we're declaring a new type **Volatile_Long_Float** based on the **Long_Float** type and using the Volatile aspect. Any object of this type is automatically volatile.

In addition to that, we can declare components of an array to be volatile. In this case, we can use the **Volatile_Components** aspect in the array declaration. For example:

[Ada]

Listing 15: show_volatile_array_components.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Show_Volatile_Array_Components is
4     Arr : array (1 .. 2) of Long_Float with Volatile_Components;
5 begin
6     Arr := (others => 0.0);
7
8     for I in 0 .. 999 loop
9         Arr (1) := Arr (1) + 2.0 * Long_Float (I);
10        Arr (2) := Arr (2) + 10.0 * Long_Float (I);
11    end loop;
12
13    Put_Line ("Arr (1): " & Long_Float'Image (Arr (1)));
14    Put_Line ("Arr (2): " & Long_Float'Image (Arr (2)));
15 end Show_Volatile_Array_Components;
```

Runtime output

```
Arr (1): 9.99000000000000E+05
Arr (2): 4.99500000000000E+06
```

Note that it's possible to use the `Volatile` aspect for the array declaration as well:

[Ada]

```
Arr : array (1 .. 2) of Long_Float with Volatile;
```

65.5.2 Atomic

An atomic object is an object that only accepts atomic reads and updates. The Ada standard specifies that "for an atomic object (including an atomic component), all reads and updates of the object as a whole are indivisible." In this case, the compiler must generate Assembly code in such a way that reads and updates of an atomic object must be done in a single instruction, so that no other instruction could execute on that same object before the read or update completes.

In other contexts

Generally, we can say that operations are said to be atomic when they can be completed without interruptions. This is an important requirement when we're performing operations on objects in memory that are shared between multiple processes.

This definition of atomicity above is used, for example, when implementing databases. However, for this section, we're using the term "atomic" differently. Here, it really means that reads and updates must be performed with a single Assembly instruction.

For example, if we have a 32-bit object composed of four 8-bit bytes, the compiler cannot generate code to read or update the object using four 8-bit store / load instructions, or even two 16-bit store / load instructions. In this case, in order to maintain atomicity, the compiler must generate code using one 32-bit store / load instruction.

Because of this strict definition, we might have objects for which the `Atomic` aspect cannot be specified. Lots of machines support integer types that are larger than the native word-sized integer. For example, a 16-bit machine probably supports both 16-bit and 32-bit integers, but only 16-bit integer objects can be marked as atomic — or, more generally, only objects that fit into at most 16 bits.

Atomicity may be important, for example, when dealing with shared hardware registers. In fact, for certain architectures, the hardware may require that memory-mapped registers are handled atomically. In Ada, we can use the `Atomic` aspect to indicate that an object is atomic. This is how we can use the aspect to declare a shared hardware register:

[Ada]

Listing 16: show_shared_hw_register.adb

```

1  with System;
2
3  procedure Show_Shared_HW_Register is
4      R : Integer
5          with Atomic, Address => System'To_Address (16#FFFF00A0#);
6  begin
7      null;
8  end Show_Shared_HW_Register;
```

Note that the `Address` aspect allows for assigning a variable to a specific location in the memory. In this example, we're using this aspect to specify the address of the memory-mapped register. We'll discuss more about the `Address` aspect later in the section about *mapping structures to bit-fields* (page 754) (in chapter 6).

In addition to atomic objects, we can declare atomic types and atomic array components — similarly to what we've seen before for volatile objects. For example:

[Ada]

Listing 17: show_shared_hw_register.adb

```

1  with System;
2
3  procedure Show_Shared_HW_Register is
4      type Atomic_Integer is new Integer with Atomic;
5
6      R : Atomic_Integer with Address => System'To_Address (16#FFFF00A0#);
7
8      Arr : array (1 .. 2) of Integer with Atomic_Components;
9  begin
10     null;
11  end Show_Shared_HW_Register;
```

In this example, we're declaring the `Atomic_Integer` type, which is an atomic type. Objects of this type — such as `R` in this example — are automatically atomic. This example also includes the declaration of the `Arr` array, which has atomic components.

65.6 Interfacing with Devices

Previously, we've seen that we can use *representation clauses* (page 702) to specify a particular layout for a record type. As mentioned before, this is useful when interfacing with hardware, drivers, or communication protocols. In this section, we'll extend this concept for two specific use-cases: register overlays and data streams. Before we discuss those use-cases, though, we'll first explain the `Size` aspect and the `Size` attribute.

65.6.1 Size aspect and attribute

The Size aspect indicates the minimum number of bits required to represent an object. When applied to a type, the Size aspect is telling the compiler to not make record or array components of a type T any smaller than X bits. Therefore, a common usage for this aspect is to just confirm expectations: developers specify '`Size`' to tell the compiler that T should fit X bits, and the compiler will tell them if they are right (or wrong).

When the specified size value is larger than necessary, it can cause objects to be bigger in memory than they would be otherwise. For example, for some enumeration types, we could say `for type Enum'Size use 32`; when the number of literals would otherwise have required only a byte. That's useful for unchecked conversions because the sizes of the two types need to be the same. Likewise, it's useful for interfacing with C, where `enum` types are just mapped to the `int` type, and thus larger than Ada might otherwise require. We'll discuss unchecked conversions *later in the course* (page 767).

Let's look at an example from an earlier chapter:

[Ada]

Listing 18: my_device_types.ads

```

1 package My_Device_Types is
2
3     type UInt10 is mod 2 ** 10
4         with Size => 10;
5
6 end My_Device_Types;
```

Here, we're saying that objects of type `UInt10` must have at least 10 bits. In this case, if the code compiles, it is a confirmation that such values can be represented in 10 bits when packed into an enclosing record or array type.

If the size specified was larger than what the compiler would use by default, then it could affect the size of objects. For example, for `UInt10`, anything up to and including 16 would make no difference on a typical machine. However, anything over 16 would then push the compiler to use a larger object representation. That would be important for unchecked conversions, for example.

The `Size` attribute indicates the number of bits required to represent a type or an object. We can use the `size` attribute to retrieve the size of a type or of an object:

[Ada]

Listing 19: show_device_types.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with My_Device_Types; use My_Device_Types;
4
5 procedure Show_Device_Types is
6     UInt10_0bj : constant UInt10 := 0;
7 begin
8     Put_Line ("Size of UInt10 type: " & Positive'Image (UInt10'Size));
9     Put_Line ("Size of UInt10 object: " & Positive'Image (UInt10_0bj'Size));
10 end Show_Device_Types;
```

Runtime output

```
Size of UInt10 type: 10
Size of UInt10 object: 16
```

Here, we're retrieving the actual sizes of the `UInt10` type and an object of that type. Note

that the sizes don't necessarily need to match. For example, although the size of `UInt10` type is expected to be 10 bits, the size of `UInt10_Obj` may be 16 bits, depending on the platform. Also, components of this type within composite types (arrays, records) will probably be 16 bits as well unless they are packed.

65.6.2 Register overlays

Register overlays make use of representation clauses to create a structure that facilitates manipulating bits from registers. Let's look at a simplified example of a power management controller containing registers such as a system clock enable register. Note that this example is based on an actual architecture:

[Ada]

Listing 20: registers.ads

```

1  with System;
2
3  package Registers is
4
5      type Bit    is mod 2 ** 1
6          with Size => 1;
7      type UInt5  is mod 2 ** 5
8          with Size => 5;
9      type UInt10 is mod 2 ** 10
10         with Size => 10;
11
12      subtype USB_Clock_Enable is Bit;
13
14      -- System Clock Enable Register
15      type PMC_SCER_Register is record
16          -- Reserved bits
17          Reserved_0_4   : UInt5           := 16#0#;
18          -- Write-only. Enable USB FS Clock
19          USBCLK        : USB_Clock_Enable := 16#0#;
20          -- Reserved bits
21          Reserved_6_15  : UInt10          := 16#0#;
22      end record
23      with
24          Volatile,
25          Size    => 16,
26          Bit_Order => System.Low_Order_First;
27
28      for PMC_SCER_Register use record
29          Reserved_0_4  at 0 range 0 .. 4;
30          USBCLK       at 0 range 5 .. 5;
31          Reserved_6_15 at 0 range 6 .. 15;
32      end record;
33
34      -- Power Management Controller
35      type PMC_Peripheral is record
36          -- System Clock Enable Register
37          PMC_SCER     : aliased PMC_SCER_Register;
38          -- System Clock Disable Register
39          PMC_SCDR     : aliased PMC_SCER_Register;
40      end record
41      with Volatile;
42
43      for PMC_Peripheral use record
44          -- 16-bit register at byte 0
45          PMC_SCER     at 16#0# range 0 .. 15;

```

(continues on next page)

(continued from previous page)

```

46      -- 16-bit register at byte 2
47      PMC_SCDR      at 16#2# range 0 .. 15;
48  end record;
49
50  -- Power Management Controller
51  PMC_Periph : aliased PMC_Peripheral
52    with Import, Address => System'To_Address (16#400E0600#);
53
54 end Registers;
```

First, we declare the system clock enable register — this is PMC_SCER_Register type in the code example. Most of the bits in that register are reserved. However, we're interested in bit #5, which is used to activate or deactivate the system clock. To achieve a correct representation of this bit, we do the following:

- We declare the USBCLK component of this record using the USB_Clock_Enable type, which has a size of one bit; and
- we use a representation clause to indicate that the USBCLK component is specifically at bit #5 of byte #0.

After declaring the system clock enable register and specifying its individual bits as components of a record type, we declare the power management controller type — PMC_Peripheral record type in the code example. Here, we declare two 16-bit registers as record components of PMC_Peripheral. These registers are used to enable or disable the system clock. The strategy we use in the declaration is similar to the one we've just seen above:

- We declare these registers as components of the PMC_Peripheral record type;
 - we use a representation clause to specify that the PMC_SCER register is at byte #0 and the PMC_SCDR register is at byte #2.
- Since these registers have 16 bits, we use a range of bits from 0 to 15.

The actual power management controller becomes accessible by the declaration of the PMC_Periph object of PMC_Peripheral type. Here, we specify the actual address of the memory-mapped registers (400E0600 in hexadecimal) using the **Address** aspect in the declaration. When we use the **Address** aspect in an object declaration, we're indicating the address in memory of that object.

Because we specify the address of the memory-mapped registers in the declaration of PMC_Periph, this object is now an overlay for those registers. This also means that any operation on this object corresponds to an actual operation on the registers of the power management controller. We'll discuss more details about overlays in the section about *mapping structures to bit-fields* (page 754) (in chapter 6).

Finally, in a test application, we can access any bit of any register of the power management controller with simple record component selection. For example, we can set the USBCLK bit of the PMC_SCER register by using PMC_Periph.PMC_SCER.USBCLK:

[Ada]

Listing 21: enable_usb_clock.adb

```

1  with Registers;
2
3  procedure Enable_USB_Clock is
4  begin
5    Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
6  end Enable_USB_Clock;
```

This code example makes use of many aspects and keywords of the Ada language. One

of them is the `Volatile` aspect, which we've discussed in the section about *volatile and atomic objects* (page 710). Using the `Volatile` aspect for the `PMC_SCER_Register` type ensures that objects of this type won't be stored in a register.

In the declaration of the `PMC_SCER_Register` record type of the example, we use the `Bit_Order` aspect to specify the bit ordering of the record type. Here, we can select one of these options:

- `High_Order_First`: first bit of the record is the most significant bit;
- `Low_Order_First`: first bit of the record is the least significant bit.

The declarations from the `Registers` package also makes use of the `Import`, which is sometimes necessary when creating overlays. When used in the context of object declarations, it avoids default initialization (for data types that have it.). Aspect `Import` will be discussed in the section that explains how to *map structures to bit-fields* (page 754) in chapter 6. Please refer to that chapter for more details.

Details about 'Size

In the example above, we're using the `Size` aspect in the declaration of the `PMC_SCER_Register` type. In this case, the effect is that it has the compiler confirm that the record type will fit into the expected 16 bits.

That's what the aspect does for type `PMC_SCER_Register` in the example above, as well as for the types `Bit`, `UInt5` and `UInt10`. For example, we may declare a stand-alone object of type `Bit`:

Listing 22: show_bit_declaration.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Bit_Declaration is
4
5    type Bit      is mod 2 ** 1
6    with Size => 1;
7
8    B : constant Bit := 0;
9    -- ^ Although Bit'Size is 1, B'Size is almost certainly 8
10 begin
11   Put_Line ("Bit'Size = " & Positive'Image (Bit'Size));
12   Put_Line ("B'Size   = " & Positive'Image (B'Size));
13 end Show_Bit_Declaration;

```

Runtime output

```

Bit'Size = 1
B'Size  = 8

```

In this case, `B` is almost certainly going to be 8-bits wide on a typical machine, even though the language requires that `Bit'Size` is 1 by default.

In the declaration of the components of the `PMC_Peripheral` record type, we use the `aliased` keyword to specify that those record components are accessible via other paths besides the component name. Therefore, the compiler won't store them in registers. This makes sense because we want to ensure that we're accessing specific memory-mapped registers, and not registers assigned by the compiler. Note that, for the same reason, we also use the `aliased` keyword in the declaration of the `PMC_Periph` object.

65.6.3 Data streams

Creating data streams — in the context of interfacing with devices — means the serialization of arbitrary information and its transmission over a communication channel. For example, we might want to transmit the content of memory-mapped registers as byte streams using a serial port. To do this, we first need to get a serialized representation of those registers as an array of bytes, which we can then transmit over the serial port.

Serialization of arbitrary record types — including register overlays — can be achieved by declaring an array of bytes as an overlay. By doing this, we're basically interpreting the information from those record types as bytes while ignoring their actual structure — i.e. their components and representation clause. We'll discuss details about overlays in the section about *mapping structures to bit-fields* (page 754) (in chapter 6).

Let's look at a simple example of serialization of an arbitrary record type:

[Ada]

Listing 23: arbitrary_types.ads

```
1 package Arbitrary_Types is
2
3     type Arbitrary_Record is record
4         A : Integer;
5         B : Integer;
6         C : Integer;
7     end record;
8
9 end Arbitrary_Types;
```

Listing 24: serialize_data.ads

```
1 with Arbitrary_Types;
2
3 procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record);
```

Listing 25: serialize_data.adb

```
1 with Arbitrary_Types;
2
3 procedure Serialize_Data (Some_Object : Arbitrary_Types.Arbitrary_Record) is
4     type UByte is new Natural range 0 .. 255
5     with Size => 8;
6
7     type UByte_Array is array (Positive range <>) of UByte;
8
9     --
10    -- We can access the serialized data in Raw_TX, which is our overlay
11    --
12    Raw_TX : UByte_Array (1 .. Some_Object'Size / 8)
13    with Address => Some_Object'Address;
14 begin
15     null;
16
17     -- Now, we could stream the data from Some_Object.
18
19     -- For example, we could send the bytes (from Raw_TX) via the
20     -- serial port.
21
22 end Serialize_Data;
```

Listing 26: data_stream_declaration.adb

```

1  with Arbitrary_Types;
2  with Serialize_Data;
3
4  procedure Data_Stream_Declaration is
5      Dummy_Object : Arbitrary_Types.Arbitrary_Record;
6
7  begin
8      Serialize_Data (Dummy_Object);
9  end Data_Stream_Declaration;

```

The most important part of this example is the implementation of the `Serialize_Data` procedure, where we declare `Raw_TX` as an overlay for our arbitrary object (`Some_Object` of `Arbitrary_Record` type). In simple terms, by writing `with Address => Some_Object'Address;` in the declaration of `Raw_TX`, we're specifying that `Raw_TX` and `Some_Object` have the same address in memory. Here, we are:

- taking the address of `Some_Object` — using the **Address** attribute —, and then
- using it as the address of `Raw_TX` — which is specified with the **Address** aspect.

By doing this, we're essentially saying that both `Raw_TX` and `Some_Object` are different representations of the same object in memory.

Because the `Raw_TX` overlay is completely agnostic about the actual structure of the record type, the `Arbitrary_Record` type could really be anything. By declaring `Raw_TX`, we create an array of bytes that we can use to stream the information from `Some_Object`.

We can use this approach and create a data stream for the register overlay example that we've seen before. This is the corresponding implementation:

[Ada]

Listing 27: registers.ads

```

1  with System;
2
3  package Registers is
4
5      type Bit is mod 2 ** 1
6          with Size => 1;
7      type UInt5 is mod 2 ** 5
8          with Size => 5;
9      type UInt10 is mod 2 ** 10
10         with Size => 10;
11
12      subtype USB_Clock_Enable is Bit;
13
14      -- System Clock Register
15      type PMC_SCER_Register is record
16          -- Reserved bits
17          Reserved_0_4    : UInt5 := 16#0#;
18          -- Write-only. Enable USB FS Clock
19          USBCLK         : USB_Clock_Enable := 16#0#;
20          -- Reserved bits
21          Reserved_6_15  : UInt10 := 16#0#;
22      end record
23      with
24          Volatile,
25          Size      => 16,
26          Bit_Order => System.Low_Order_First;
27

```

(continues on next page)

(continued from previous page)

```

28   for PMC_SCER_Register use record
29     Reserved_0_4  at 0 range 0 .. 4;
30     USBCLK       at 0 range 5 .. 5;
31     Reserved_6_15 at 0 range 6 .. 15;
32   end record;
33
34   -- Power Management Controller
35   type PMC_Peripheral is record
36     -- System Clock Enable Register
37     PMC_SCER      : aliased PMC_SCER_Register;
38     -- System Clock Disable Register
39     PMC_SCDR      : aliased PMC_SCER_Register;
40   end record
41   with Volatile;
42
43   for PMC_Peripheral use record
44     -- 16-bit register at byte 0
45     PMC_SCER      at 16#0# range 0 .. 15;
46     -- 16-bit register at byte 2
47     PMC_SCDR      at 16#2# range 0 .. 15;
48   end record;
49
50   -- Power Management Controller
51   PMCPeriph : aliased PMC_Peripheral;
52   -- with Import, Address => System'To_Address (16#400E0600#);
53
54 end Registers;

```

Listing 28: serial_ports.ads

```

1 package Serial_Ports is
2
3   type UByte is new Natural range 0 .. 255
4     with Size => 8;
5
6   type UByte_Array is array (Positive range <>) of UByte;
7
8   type Serial_Port is null record;
9
10  procedure Read (Port : in out Serial_Port;
11                  Data :        out UByte_Array);
12
13  procedure Write (Port : in out Serial_Port;
14                    Data :           UByte_Array);
15
16 end Serial_Ports;

```

Listing 29: serial_ports.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Serial_Ports is
4
5   procedure Display (Data : UByte_Array) is
6     begin
7       Put_Line ("---- Data ----");
8       for E of Data loop
9         Put_Line (UByte'Image (E));
10      end loop;
11      Put_Line ("-----");

```

(continues on next page)

(continued from previous page)

```

12  end Display;
13
14  procedure Read (Port : in out Serial_Port;
15      Data :    out UByte_Array) is
16      pragma Unreferenced (Port);
17  begin
18      Put_Line ("Reading data...");
19      Data := (0, 0, 32, 0);
20  end Read;
21
22  procedure Write (Port : in out Serial_Port;
23      Data :        UByte_Array) is
24      pragma Unreferenced (Port);
25  begin
26      Put_Line ("Writing data...");
27      Display (Data);
28  end Write;
29
30 end Serial_Ports;

```

Listing 30: data_stream.ads

```

1  with Serial_Ports; use Serial_Ports;
2  with Registers;   use Registers;
3
4  package Data_Stream is
5
6      procedure Send (Port : in out Serial_Port;
7                      PMC  :        PMC_Peripheral);
8
9      procedure Receive (Port : in out Serial_Port;
10                     PMC  :        out PMC_Peripheral);
11
12 end Data_Stream;

```

Listing 31: data_stream.adb

```

1  package body Data_Stream is
2
3      procedure Send (Port : in out Serial_Port;
4                      PMC  :        PMC_Peripheral)
5  is
6          Raw_TX : UByte_Array (1 .. PMC'Size / 8)
7          with Address => PMC'Address;
8      begin
9          Write (Port => Port,
10                 Data => Raw_TX);
11      end Send;
12
13      procedure Receive (Port : in out Serial_Port;
14                         PMC  :        out PMC_Peripheral)
15  is
16          Raw_TX : UByte_Array (1 .. PMC'Size / 8)
17          with Address => PMC'Address;
18      begin
19          Read (Port => Port,
20                 Data => Raw_TX);
21      end Receive;
22
23 end Data_Stream;

```

Listing 32: test_data_stream.adb

```
1  with Ada.Text_IO;
2
3  with Registers;
4  with Data_Stream;
5  with Serial_Ports;
6
7  procedure Test_Data_Stream is
8
9    procedure Display_Registers is
10      use Ada.Text_IO;
11    begin
12      Put_Line ("---- Registers ----");
13      Put_Line ("PMC_SCER.USBCLK: "
14                  & Registers.PMC_Periph.PMC_SCER.USBCLK'Image);
15      Put_Line ("PMC_SCDR.USBCLK: "
16                  & Registers.PMC_Periph.PMC_SCDR.USBCLK'Image);
17      Put_Line ("----- -----");
18    end Display_Registers;
19
20    Port : Serial_Ports.Serial_Port;
21  begin
22    Registers.PMC_Periph.PMC_SCER.USBCLK := 1;
23    Registers.PMC_Periph.PMC_SCDR.USBCLK := 1;
24
25    Display_Registers;
26
27    Data_Stream.Send (Port => Port,
28                      PMC  => Registers.PMC_Periph);
29
30    Data_Stream.Receive (Port => Port,
31                      PMC  => Registers.PMC_Periph);
32
33    Display_Registers;
34  end Test_Data_Stream;
```

Runtime output

```
---- Registers ----
PMC_SCER.USBCLK: 1
PMC_SCDR.USBCLK: 1
-----
Writing data...
---- Data ----
32
0
32
0
-----
Reading data...
---- Registers ----
PMC_SCER.USBCLK: 0
PMC_SCDR.USBCLK: 1
-----
```

In this example, we can find the overlay in the implementation of the Send and Receive procedures from the Data_Stream package. Because the overlay doesn't need to know the internals of the PMC_Peripheral type, we're declaring it in the same way as in the previous example (where we created an overlay for Some_Object). In this case, we're creating an overlay for the PMC parameter.

Note that, for this section, we're not really interested in the details about the serial port. Thus, package `Serial_Ports` in this example is just a stub. However, because the `Serial_Port` type in that package only sees arrays of bytes, after implementing an actual serial port interface for a specific device, we could create data streams for any type.

65.7 ARM and svd2ada

As we've seen in the previous section about *interfacing with devices* (page 713), Ada offers powerful features to describe low-level details about the hardware architecture without giving up its strong typing capabilities. However, it can be cumbersome to create a specification for all those low-level details when you have a complex architecture. Fortunately, for ARM Cortex-M devices, the GNAT toolchain offers an Ada binding generator called **svd2ada**, which takes CMSIS-SVD descriptions for those devices and creates Ada specifications that match the architecture. CMSIS-SVD description files are based on the Cortex Microcontroller Software Interface Standard (CMSIS), which is a hardware abstraction layer for ARM Cortex microcontrollers.

Please refer to the `svd2ada` project page¹⁰⁶ for details about this tool.

¹⁰⁶ <https://github.com/AdaCore/svd2ada>

ENHANCING VERIFICATION WITH SPARK AND ADA

66.1 Understanding Exceptions and Dynamic Checks

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering "exceptions" that interrupt the normal flow of execution. For example, an exception is raised by an attempt to access an array component via an index that is out of bounds. This simple check precludes exploits based on buffer overflow. Several other cases also raise language-defined exceptions, such as scalar range constraint violations and null pointer dereferences. Developers may declare and raise their own application-specific exceptions too. (Exceptions are software artifacts, although an implementation may map hardware events to exceptions.)

Exceptions are raised during execution of what we will loosely define as a "frame." A frame is a language construct that has a call stack entry when called, for example a procedure or function body. There are a few other constructs that are also pertinent but this definition will suffice for now.

Frames have a sequence of statements implementing their functionality. They can also have optional "exception handlers" that specify the response when exceptions are "raised" by those statements. These exceptions could be raised directly within the statements, or indirectly via calls to other procedures and functions.

For example, the frame below is a procedure including three exceptions handlers:

Listing 1: p.adb

```
1  procedure P is
2  begin
3      Statements_That_Might_Raise_Exceptions;
4  exception
5      when A =>
6          Handle_A;
7      when B =>
8          Handle_B;
9      when C =>
10         Handle_C;
11 end P;
```

The three exception handlers each start with the word `when` (lines 5, 7, and 9). Next comes one or more exception identifiers, followed by the so-called "arrow." In Ada, the arrow always associates something on the left side with something on the right side. In this case, the left side is the exception name and the right side is the handler's code for that exception.

Each handler's code consists of an arbitrary sequence of statements, in this case specific procedures called in response to those specific exceptions. If exception A is raised we call procedure `Handle_A` (line 6), dedicated to doing the actual work of handling that exception. The other two exceptions are dealt with similarly, on lines 8 and 10.

Structurally, the exception handlers are grouped together and textually separated from the rest of the code in a frame. As a result, the sequence of statements representing the normal flow of execution is distinct from the section representing the error handling. The reserved word **exception** separates these two sections (line 4 above). This separation helps simplify the overall flow, increasing understandability. In particular, status result codes are not required so there is no mixture of error checking and normal processing. If no exception is raised the exception handler section is automatically skipped when the frame exits.

Note how the syntactic structure of the exception handling section resembles that of an Ada case statement. The resemblance is intentional, to suggest similar behavior. When something in the statements of the normal execution raises an exception, the corresponding exception handler for that specific exception is executed. After that, the routine completes. The handlers do not "fall through" to the handlers below. For example, if exception B is raised, procedure Handle_B is called but Handle_C is not called. There's no need for a **break** statement, just as there is no need for it in a case statement. (There's no break statement in Ada anyway.)

So far, we've seen a frame with three specific exceptions handled. What happens if a frame has no handler for the actual exception raised? In that case the run-time library code goes "looking" for one.

Specifically, the active exception is propagated up the dynamic call chain. At each point in the chain, normal execution in that caller is abandoned and the handlers are examined. If that caller has a handler for the exception, the handler is executed. That caller then returns normally to its caller and execution continues from there. Otherwise, propagation goes up one level in the call chain and the process repeats. The search continues until a matching handler is found or no callers remain. If a handler is never found the application terminates abnormally. If the search reaches the main procedure and it has a matching handler it will execute the handler, but, as always, the routine completes so once again the application terminates.

For a concrete example, consider the following:

Listing 2: arrays.ads

```
1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Listing 3: arrays.adb

```
1 package body Arrays is
2
3     function Value (A : List; X, Y : Integer) return Integer is
4         begin
5             return A (X + Y * 10);
6         end Value;
7
8 end Arrays;
```

Listing 4: some_process.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
```

(continues on next page)

(continued from previous page)

```

5   L : constant List (1 .. 100) := (others => 42);
6 begin
7   Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9   when Constraint_Error =>
10      Put_Line ("Constraint_Error caught in Some_Process");
11      Put_Line ("Some_Process completes normally");
12 end Some_Process;

```

Listing 5: main.adb

```

1 with Some_Process;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Main is
5 begin
6   Some_Process;
7   Put_Line ("Main completes normally");
8 end Main;

```

Procedure Main calls Some_Process, which in turn calls function Value (line 7). Some_Process declares the array object L of type List on line 5, with bounds 1 through 100. The call to Value has arguments, including variable L, leading to an attempt to access an array component via an out-of-bounds index ($1 + 10 * 10 = 101$, beyond the last index of L). This attempt will trigger an exception in Value prior to actually accessing the array object's memory. Function Value doesn't have any exception handlers so the exception is propagated up to the caller Some_Process. Procedure Some_Process has an exception handler for Constraint_Error and it so happens that Constraint_Error is the exception raised in this case. As a result, the code for that handler will be executed, printing some messages on the screen. Then procedure Some_Process will return to Main normally. Main then continues to execute normally after the call to Some_Process and prints its completion message.

If procedure Some_Process had also not had a handler for Constraint_Error, that procedure call would also have returned abnormally and the exception would have been propagated further up the call chain to procedure Main. Normal execution in Main would likewise be abandoned in search of a handler. But Main does not have any handlers so Main would have completed abnormally, immediately, without printing its closing message.

This semantic model is the same as with many other programming languages, in which the execution of a frame's sequence of statements is unavoidably abandoned when an exception becomes active. The model is a direct reaction to the use of status codes returned from functions as in C, where it is all too easy to forget (intentionally or otherwise) to check the status values returned. With the exception model errors cannot be ignored.

However, full exception propagation as described above is not the norm for embedded applications when the highest levels of integrity are required. The run-time library code implementing exception propagation can be rather complex and expensive to certify. Those problems apply to the application code too, because exception propagation is a form of control flow without any explicit construct in the source. Instead of the full exception model, designers of high-integrity applications often take alternative approaches.

One alternative consists of deactivating exceptions altogether, or more precisely, deactivating language-defined checks, which means that the compiler will not generate code checking for conditions giving rise to exceptions. Of course, this makes the code vulnerable to attacks, such as buffer overflow, unless otherwise verified (e.g. through static analysis). Deactivation can be applied at the unit level, through the -gnatp compiler switch, or locally within a unit via the pragma Suppress. (Refer to the [GNAT User's Guide for Native Platforms](#)¹⁰⁷ for more details about the switch.)

¹⁰⁷ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_-gnatp.html

For example, we can write the following. Note the pragma on line 4 of arrays.adb within function Value:

Listing 6: arrays.ads

```
1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Listing 7: arrays.adb

```
1 package body Arrays is
2
3     function Value (A : List; X, Y : Integer) return Integer is
4         pragma Suppress (All_Checks);
5     begin
6         return A (X + Y * 10);
7     end Value;
8
9 end Arrays;
```

Listing 8: some_process.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
5     L : constant List (1 .. 100) := (others => 42);
6 begin
7     Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9     when Constraint_Error =>
10        Put_Line ("FAILURE");
11 end Some_Process;
```

This placement of the pragma will only suppress checks in the function body. However, that is where the exception would otherwise have been raised, leading to incorrect and unpredictable execution. (Run the program more than once. If it prints the right answer (42), or even the same value each time, it's just a coincidence.) As you can see, suppressing checks negates the guarantee of errors being detected and addressed at run-time.

Another alternative is to leave checks enabled but not retain the dynamic call-chain propagation. There are a couple of approaches available in this alternative.

The first approach is for the run-time library to invoke a global "last chance handler" (LCH) when any exception is raised. Instead of the sequence of statements of an ordinary exception handler, the LCH is actually a procedure intended to perform "last-wishes" before the program terminates. No exception handlers are allowed. In this scheme "propagation" is simply a direct call to the LCH procedure. The default LCH implementation provided by GNAT does nothing other than loop infinitely. Users may define their own replacement implementation.

The availability of this approach depends on the run-time library. Typically, *Zero Footprint* and *Ravenscar SFP* run-times will provide this mechanism because they are intended for certification.

A user-defined LCH handler can be provided either in C or in Ada, with the following profiles:

gnat.html

[Ada]

```
procedure Last_Chance_Handler (Source_Location : System.Address; Line : Integer);
pragma Export (C,
    Last_Chance_Handler,
    "__gnat_last_chance_handler");
```

[C]

```
void __gnat_last_chance_handler (char *source_location,
                                 int line);
```

We'll go into the details of the pragma Export in a further section on language interfacing. For now, just know that the symbol `__gnat_last_chance_handler` is what the run-time uses to branch immediately to the last-chance handler. Pragma Export associates that symbol with this replacement procedure so it will be invoked instead of the default routine. As a consequence, the actual procedure name in Ada is immaterial.

Here is an example implementation that simply blinks an LED forever on the target:

```
procedure Last_Chance_Handler (Msg : System.Address; Line : Integer) is
    pragma Unreferenced (Msg, Line);

    Next_Release : Time := Clock;
    Period       : constant Time_Span := Milliseconds (500);
begin
    Initialize_LEDs;
    All_LEDs_Off;

    loop
        Toggle (LCH_LED);
        Next_Release := Next_Release + Period;
        delay until Next_Release;
    end loop;
end Last_Chance_Handler;
```

The `LCH_LED` is a constant referencing the LED used by the last-chance handler, declared elsewhere. The infinite loop is necessary because a last-chance handler must never return to the caller (hence the term "last-chance"). The LED changes state every half-second.

Unlike the approach in which there is only the last-chance handler routine, the other approach allows exception handlers, but in a specific, restricted manner. Whenever an exception is raised, the only handler that can apply is a matching handler located in the same frame in which the exception is raised. Propagation in this context is simply an immediate branch instruction issued by the compiler, going directly to the matching handler's sequence of statements. If there is no matching local handler the last chance handler is invoked. For example consider the body of function `Value` in the body of package `Arrays`:

Listing 9: arrays.ads

```
1 package Arrays is
2
3     type List is array (Natural range <>) of Integer;
4
5     function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;
```

Listing 10: arrays.adb

```
1 package body Arrays is
2
```

(continues on next page)

(continued from previous page)

```

3   function Value (A : List; X, Y : Integer) return Integer is
4     begin
5       return A (X + Y * 10);
6     exception
7       when Constraint_Error =>
8         return 0;
9     end Value;
10
11 end Arrays;

```

Listing 11: some_process.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
5   L : constant List (1 .. 100) := (others => 42);
6 begin
7   Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9   when Constraint_Error =>
10    Put_Line ("FAILURE");
11 end Some_Process;

```

In both procedure `Some_Process` and function `Value` we have an exception handler for `Constraint_Error`. In this example the exception is raised in `Value` because the index check fails there. A local handler for that exception is present so the handler applies and the function returns zero, normally. Because the call to the function returns normally, the execution of `Some_Process` prints zero and then completes normally.

Let's imagine, however, that function `Value` did *not* have a handler for `Constraint_Error`. In the context of full exception propagation, the function call would return to the caller, i.e., `Some_Process`, and would be handled in that procedure's handler. But only local handlers are allowed under the second alternative so the lack of a local handler in `Value` would result in the last-chance handler being invoked. The handler for `Constraint_Error` in `Some_Process` under this alternative approach.

So far we've only illustrated handling the `Constraint_Error` exception. It's possible to handle other language-defined and user-defined exceptions as well, of course. It is even possible to define a single handler for all other exceptions that might be encountered in the handled sequence of statements, beyond those explicitly named. The "name" for this otherwise anonymous exception is the Ada reserved word `others`. As in case statements, it covers all other choices not explicitly mentioned, and so must come last. For example:

Listing 12: arrays.ads

```

1 package Arrays is
2
3   type List is array (Natural range <>) of Integer;
4
5   function Value (A : List; X, Y : Integer) return Integer;
6
7 end Arrays;

```

Listing 13: arrays.adb

```

1 package body Arrays is
2
3   function Value (A : List; X, Y : Integer) return Integer is
4     begin

```

(continues on next page)

(continued from previous page)

```

5   return A (X + Y * 10);
6 exception
7   when Constraint_Error =>
8     return 0;
9   when others =>
10    return -1;
11 end Value;
12
13 end Arrays;

```

Listing 14: some_process.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2 with Arrays;      use Arrays;
3
4 procedure Some_Process is
5   L : constant List (1 .. 100) := (others => 42);
6 begin
7   Put_Line (Integer'Image (Value (L, 1, 10)));
8 exception
9   when Constraint_Error =>
10    Put_Line ("FAILURE");
11 end Some_Process;

```

In the code above, the Value function has a handler specifically for Constraint_Error as before, but also now has a handler for all other exceptions. For any exception other than Constraint_Error, function Value returns -1. If you remove the function's handler for Constraint_Error (lines 7 and 8) then the other "anonymous" handler will catch the exception and -1 will be returned instead of zero.

There are additional capabilities for exceptions, but for now you have a good basic understanding of how exceptions work, especially their dynamic nature at run-time.

66.2 Understanding Dynamic Checks versus Formal Proof

So far, we have discussed language-defined checks inserted by the compiler for verification at run-time, leading to exceptions being raised. We saw that these dynamic checks verified semantic conditions ensuring proper execution, such as preventing writing past the end of a buffer, or exceeding an application-specific integer range constraint, and so on. These checks are defined by the language because they apply generally and can be expressed in language-defined terms.

Developers can also define dynamic checks. These checks specify component-specific or application-specific conditions, expressed in terms defined by the component or application. We will refer to these checks as "user-defined" for convenience. (Be sure you understand that we are not talking about user-defined *exceptions* here.)

Like the language-defined checks, user-defined checks must be true at run-time. All checks consist of Boolean conditions, which is why we can refer to them as assertions: their conditions are asserted to be true by the compiler or developer.

Assertions come in several forms, some relatively low-level, such as a simple pragma Assert, and some high-level, such as type invariants and contracts. These forms will be presented in detail in a later section, but we will illustrate some of them here.

User-defined checks can be enabled at run-time in GNAT with the -gnata switch, as well as with pragma Assertion_Policy. The switch enables all forms of these assertions, whereas

the pragma can be used to control specific forms. The switch is typically used but there are reasonable use-cases in which some user-defined checks are enabled, and others, although defined, are disabled.

By default in GNAT, language-defined checks are enabled but user-defined checks are disabled. Here's an example of a simple program employing a low-level assertion. We can use it to show the effects of the switches, including the defaults:

Listing 15: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     X : Positive := 10;
5 begin
6     X := X * 5;
7     pragma Assert (X > 99);
8     X := X - 99;
9     Put_Line (Integer'Image (X));
10 end Main;
```

If we compiled this code we would get a warning about the assignment on line 8 after the pragma Assert, but not one about the Assert itself on line 7.

```
gprbuild -q -P main.gpr
main.adb:8:11: warning: value not in range of type "Standard.Positive"
main.adb:8:11: warning: "Constraint_Error" will be raised at run time
```

No code is generated for the user-defined check expressed via pragma Assert but the language-defined check is emitted. In this case the range constraint on X excludes zero and negative numbers, but $X * 5 = 50$, $X - 99 = -49$. As a result, the check for the last assignment would fail, raising Constraint_Error when the program runs. These results are the expected behavior for the default switch settings.

But now let's enable user-defined checks and build it. Different compiler output will appear.

Listing 16: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     X : Positive := 10;
5 begin
6     X := X * 5;
7     pragma Assert (X > 99);
8     X := X - 99;
9     Put_Line (Integer'Image (X));
10 end Main;
```

Build output

```
main.adb:7:19: warning: assertion will fail at run time [-gnatw.a]
main.adb:7:21: warning: condition can only be True if invalid values present [-
    ↪gnatwc]
main.adb:8:11: warning: value not in range of type "Standard.Positive" [enabled by
    ↪default]
main.adb:8:11: warning: Constraint_Error will be raised at run time [enabled by
    ↪default]
```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : main.adb:7
```

Now we also get the compiler warning about the pragma Assert condition. When run, the failure of pragma Assert on line 7 raises the exception Ada.Assertions.Assertion_Error. According to the expression in the assertion, X is expected (incorrectly) to be above 99 after the multiplication. (The exception name in the error message, SYSTEM ASSERTIONS ASSERT_FAILURE, is a GNAT-specific alias for Ada.Assertions.Assertion_Error.)

It's interesting to see in the output that the compiler can detect some violations at compile-time:

```
main.adb:7:19: warning: assertion will fail at run time
main.adb:7:21: warning: condition can only be True if invalid values present
main.adb:8:11: warning: value not in range of type "Standard.Positive"
```

Generally speaking, a complete analysis is beyond the scope of compilers and they may not find all errors prior to execution, even those we might detect ourselves by inspection. More errors can be found by tools dedicated to that purpose, known as static analyzers. But even an automated static analysis tool cannot guarantee it will find all potential problems.

A much more powerful alternative is formal proof, a form of static analysis that can (when possible) give strong guarantees about the checks, for all possible conditions and all possible inputs. Proof can be applied to both language-defined and user-defined checks.

Be sure you understand that formal proof, as a form of static analysis, verifies conditions prior to execution, even prior to compilation. That earliness provides significant cost benefits. Removing bugs earlier is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after deployment. Preventing bug introduction into the deployed system is the least expensive approach of all. Furthermore, cost savings during the initial development will be possible as well, for reasons specific to proof. We will revisit this topic later in this section.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the **gnatprove** verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof. As a disclaimer, this course is not aimed at providing a full introduction to proof and the SPARK language, but rather to present in a few examples what it is about and what it can do for us.

As it turns out, our procedure Main is already SPARK compliant so we can start verifying it.

Listing 17: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4     X : Positive := 10;
5 begin
6     X := X * 5;
7     pragma Assert (X > 99);
8     X := X - 99;
9     Put_Line (Integer'Image (X));
10 end Main;
```

Build output

```
main.adb:7:20: warning: assertion will fail at run time [-gnatw.a]
main.adb:7:22: warning: condition can only be True if invalid values present [-gnatwc]
main.adb:8:12: warning: value not in range of type "Standard.Positive" [enabled by default]
main.adb:8:12: warning: Constraint_Error will be raised at run time [enabled by default]
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
main.adb:7:20: medium: assertion might fail
gnatprove: unproved check messages considered as errors
```

Runtime output

```
raised ADA ASSERTIONS ASSERTION_ERROR : main.adb:7
```

The "Prove" button invokes **gnatprove** on main.adb. You can ignore the parameters to the invocation. For the purpose of this demonstration, the interesting output is this message:

```
main.adb:7:19: medium: assertion might fail, cannot prove X > 99 (e.g. when X = 50)
```

gnatprove can tell that the assertion $X > 99$ may have a problem. There's indeed a bug here, and **gnatprove** even gives us the counterexample (when X is 50). As a result the code is not proven and we know we have an error to correct.

Notice that the message says the assertion "might fail" even though clearly **gnatprove** has an example for when failure is certain. That wording is a reflection of the fact that SPARK gives strong guarantees when the assertions are proven to hold, but does not guarantee that flagged problems are indeed problems. In other words, **gnatprove** does not give false positives but false negatives are possible. The result is that if **gnatprove** does not indicate a problem for the code under analysis we can be sure there is no problem, but if **gnatprove** does indicate a problem the tool may be wrong.

66.3 Initialization and Correct Data Flow

An immediate benefit from having our code compatible with the SPARK subset is that we can ask **gnatprove** to verify initialization and correct data flow, as indicated by the absence of messages during SPARK "flow analysis." Flow analysis detects programming errors such as reading uninitialized data, problematic aliasing between formal parameters, and data races between concurrent tasks.

In addition, **gnatprove** checks unit specifications for the actual data read or written, and the flow of information from inputs to outputs. As you can imagine, this verification provides significant benefits, and it can be reached with comparatively low cost.

For example, the following illustrates an initialization failure:

Listing 18: main.adb

```
1 with Increment;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 procedure Main is
5   B : Integer;
6 begin
7   Increment (B);
8   Put_Line (B'Image);
9 end Main;
```

Listing 19: increment.adb

```
1 procedure Increment (Value : in out Integer) is
2 begin
3   Value := Value + 1;
4 end Increment;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
main.adb:7:15: warning: "B" may be referenced before it has a value [enabled by ← default]
main.adb:7:15: high: "B" is not initialized
gnatprove: unproved check messages considered as errors
```

Granted, Increment is a silly procedure as-is, but imagine it did useful things, and, as part of that, incremented the argument. **gnatprove** tells us that the caller has not assigned a value to the argument passed to Increment.

Consider this next routine, which contains a serious coding error. Flow analysis will find it for us.

Listing 20: compute_offset.adb

```
1  with Ada.Numerics.Elementary_Functions;  use Ada.Numerics.Elementary_Functions;
2
3  procedure Compute_Offset (K : Float; Z : out Integer; Flag : out Boolean) is
4      X : constant Float := Sin (K);
5  begin
6      if X < 0.0 then
7          Z := 0;
8          Flag := True;
9      elsif X > 0.0 then
10         Z := 1;
11         Flag := True;
12     else
13         Flag := False;
14     end if;
15  end Compute_Offset;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
compute_offset.adb:3:38: medium: "Z" might not be initialized in "Compute_Offset" \[reason for check: OUT parameter should be initialized on return\] \[possible fix: ← initialize "Z" on all paths or make "Z" an IN OUT parameter\]
gnatprove: unproved check messages considered as errors
```

gnatprove tells us that Z might not be initialized (assigned a value) in Compute_Offset, and indeed that is correct. Z is a mode **out** parameter so the routine should assign a value to it: Z is an output, after all. The fact that Compute_Offset does not do so is a significant and nasty bug. Why is it so nasty? In this case, formal parameter Z is of the scalar type **Integer**, and scalar parameters are always passed by copy in Ada and SPARK. That means that, when returning to the caller, an integer value is copied to the caller's argument passed to Z. But this procedure doesn't always assign the value to be copied back, and in that case an arbitrary value — whatever is on the stack — is copied to the caller's argument. The poor programmer must debug the code to find the problem, yet the effect could appear well downstream from the call to Compute_Offset. That's not only painful, it is expensive. Better to find the problem before we even compile the code.

66.4 Contract-Based Programming

So far, we've seen assertions in a routine's sequence of statements, either through implicit language-defined checks (is the index in the right range?) or explicit user-defined checks. These checks are already useful by themselves but they have an important limitation: the assertions are in the implementation, hidden from the callers of the routine. For example, a call's success or failure may depend upon certain input values but the caller doesn't have that information.

Generally speaking, Ada and SPARK put a lot of emphasis on strong, complete specifications for the sake of abstraction and analysis. Callers need not examine the implementations to determine whether the arguments passed to it are changed, for example. It is possible to go beyond that, however, to specify implementation constraints and functional requirements. We use contracts to do so.

At the language level, contracts are higher-level forms of assertions associated with specifications and declarations rather than sequences of statements. Like other assertions they can be activated or deactivated at run-time, and can be statically proven. We'll concentrate here on two kinds of contracts, both associated especially (but not exclusively) with procedures and functions:

- *Preconditions*, those Boolean conditions required to be true *prior* to a call of the corresponding subprogram
- *Postconditions*, those Boolean conditions required to be true *after* a call, as a result of the corresponding subprogram's execution

In particular, preconditions specify the initial conditions, if any, required for the called routine to correctly execute. Postconditions, on the other hand, specify what the called routine's execution must have done, at least, on normal completion. Therefore, preconditions are obligations on callers (referred to as "clients") and postconditions are obligations on implementers. By the same token, preconditions are guarantees to the implementers, and postconditions are guarantees to clients.

Contract-based programming, then, is the specification and rigorous enforcement of these obligations and guarantees. Enforcement is rigorous because it is not manual, but tool-based: dynamically at run-time with exceptions, or, with SPARK, statically, prior to build.

Preconditions are specified via the "Pre" aspect. Postconditions are specified via the "Post" aspect. Usually subprograms have separate declarations and these aspects appear with those declarations, even though they are *about* the bodies. Placement on the declarations allows the obligations and guarantees to be visible to all parties. For example:

Listing 21: mid.ads

```

1  function Mid (X, Y : Integer) return Integer with
2    Pre => X + Y /= 0,
3    Post => Mid'Result > X;

```

The precondition on line 2 specifies that, for any given call, the sum of the values passed to parameters X and Y must not be zero. (Perhaps we're dividing by X + Y in the body.) The declaration also provides a guarantee about the function call's result, via the postcondition on line 3: for any given call, the value returned will be greater than the value passed to X.

Consider a client calling this function:

Listing 22: demo.adb

```

1  with Mid;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Demo is

```

(continues on next page)

(continued from previous page)

```

5   A, B, C : Integer;
6 begin
7   A := Mid (1, 2);
8   B := Mid (1, -1);
9   C := Mid (A, B);
10  Put_Line (C'Image);
11 end Demo;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
demo.adb:8:09: medium: precondition might fail
gnatprove: unproved check messages considered as errors

```

gnatprove indicates that the assignment to B (line 8) might fail because of the precondition, i.e., the sum of the inputs shouldn't be 0, yet $-1 + 1 = 0$. (We will address the other output message elsewhere.)

Let's change the argument passed to Y in the second call (line 8). Instead of -1 we will pass -2:

Listing 23: demo.adb

```

1  with Mid;
2  with Ada.Text_IO; use Ada.Text_IO;
3
4  procedure Demo is
5    A, B, C : Integer;
6  begin
7    A := Mid (1, 2);
8    B := Mid (1, -2);
9    C := Mid (A, B);
10   Put_Line (C'Image);
11 end Demo;

```

Listing 24: mid.ads

```

1  function Mid (X, Y : Integer) return Integer with
2    Pre  => X + Y /= 0,
3    Post => Mid'Result > X;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
warning: no bodies have been analyzed by GNATprove
enable analysis of a non-generic body using SPARK_Mode

```

The second call will no longer be flagged for the precondition. In addition, **gnatprove** will know from the postcondition that A has to be greater than 1, as does B, because in both calls 1 was passed to X. Therefore, **gnatprove** can deduce that the precondition will hold for the third call $C := Mid (A, B)$; because the sum of two numbers greater than 1 will never be zero.

Postconditions can also compare the state prior to a call with the state after a call, using the '**Old**' attribute. For example:

Listing 25: increment.ads

```

1 procedure Increment (Value : in out Integer) with
2   Pre => Value < Integer'Last,
3   Post => Value = Value'Old + 1;

```

Listing 26: increment.adb

```

1 procedure Increment (Value : in out Integer) is
2 begin
3   Value := Value + 1;
4 end Increment;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...

```

The postcondition specifies that, on return, the argument passed to the parameter `Value` will be one greater than it was immediately prior to the call (`Value'Old`).

66.5 Replacing Defensive Code

One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations. For example, the `Push` operation for a stack type would need to ensure that the given stack is not already full. The body of the routine would first check that, explicitly, and perhaps raise an exception or set a status code. With preconditions we can make the requirement explicit and `gnatprove` will verify that the requirement holds at all call sites.

This reduction has a number of advantages:

- The implementation is simpler, removing validation code that is often difficult to test, makes the code more complex and leads to behaviors that are difficult to define.
- The precondition documents the conditions under which it's correct to call the subprogram, moving from an implementer responsibility to mitigate invalid input to a user responsibility to fulfill the expected interface.
- Provides the means to verify that this interface is properly respected, through code review, dynamic checking at run-time, or formal static proof.

As an example, consider a procedure `Read` that returns a component value from an array. Both the `Data` and `Index` are objects visible to the procedure so they are not formal parameters.

Listing 27: p.ads

```

1 package P is
2
3   type List is array (Integer range <>) of Character;
4
5   Data : List (1 .. 100);
6   Index : Integer := Data'First;
7
8   procedure Read (V : out Character);
9
10 end P;

```

Listing 28: p.adb

```

1 package body P is
2
3   procedure Read (V : out Character) is
4   begin
5     if Index not in Data'Range then
6       V := Character'First;
7       return;
8     end if;
9
10    V := Data (Index);
11    Index := Index + 1;
12  end Read;
13 end P;

```

Prover output

Phase 1 of 2: generation of Global contracts ...
 Phase 2 of 2: flow analysis and proof ...

In addition to procedure Read we would also have a way to load the array components in the first place, but we can ignore that for the purpose of this discussion.

Procedure Read is responsible for reading an element of the array and then incrementing the index. What should it do in case of an invalid index? In this implementation there is defensive code that returns a value arbitrarily chosen. We could also redesign the code to return a status in this case, or — better — raise an exception.

An even more robust approach would be instead to ensure that this subprogram is only called when Index is within the indexing boundaries of Data. We can express that requirement with a precondition (line 9).

Listing 29: p.ads

```

1 package P is
2
3   type List is array (Integer range <>) of Character;
4
5   Data : List (1 .. 100);
6   Index : Integer := 1;
7
8   procedure Read (V : out Character)
9     with Pre => Index in Data'Range;
10
11 end P;

```

Listing 30: p.adb

```

1 package body P is
2
3   procedure Read (V : out Character) is
4   begin
5     V := Data (Index);
6     Index := Index + 1;
7   end Read;
8
9 end P;

```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
```

Now we don't need the defensive code in the procedure body. That's safe because SPARK will attempt to prove statically that the check will not fail at the point of each call.

Assuming that procedure Read is intended to be the only way to get values from the array, in a real application (where the principles of software engineering apply) we would take advantage of the compile-time visibility controls that packages offer. Specifically, we would move all the variables' declarations to the private part of the package, or even the package body, so that client code could not possibly access the array directly. Only procedure Read would remain visible to clients, thus remaining the only means of accessing the array. However, that change would entail others, and in this chapter we are only concerned with introducing the capabilities of SPARK. Therefore, we keep the examples as simple as possible.

66.6 Proving Absence of Run-Time Errors

Earlier we said that **gnatprove** will verify both language-defined and user-defined checks. Proving that the language-defined checks will not raise exceptions at run-time is known as proving "Absence of Run-Time Errors" or AoRTE for short. Successful proof of these checks is highly significant in itself.

One of the major resulting benefits is that we can deploy the final executable with checks disabled. That has obvious performance benefits, but it is also a safety issue. If we disable the checks we also disable the run-time library support for them, but in that case the language does not define what happens if indeed an exception is raised. Formally speaking, anything could happen. We must have good reason for thinking that exceptions cannot be raised.

This is such an important issue that proof of AoRTE can be used to comply with the objectives of certification standards in various high-integrity domains (for example, DO-178B/C in avionics, EN 50128 in railway, IEC 61508 in many safety-related industries, ECSS-Q-ST-80C in space, IEC 60880 in nuclear, IEC 62304 in medical, and ISO 26262 in automotive).

As a result, the quality of the program can be guaranteed to achieve higher levels of integrity than would be possible in other programming languages.

However, successful proof of AoRTE may require additional assertions, especially preconditions. We can see that with procedure Increment, the procedure that takes an Integer argument and increments it by one. But of course, if the incoming value of the argument is the largest possible positive value, the attempt to increment it would overflow, raising Constraint_Error. (As you have likely already concluded, Constraint_Error is the most common exception you will have to deal with.) We added a precondition to allow only the integer values up to, but not including, the largest positive value:

Listing 31: increment.ads

```
1 procedure Increment (Value : in out Integer) with
2   Pre => Value < Integer'Last,
3   Post => Value = Value'Old + 1;
```

Listing 32: increment.adb

```
1 procedure Increment (Value : in out Integer) is
2 begin
3   Value := Value + 1;
4 end Increment;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
```

Prove it, then comment-out the precondition and try proving it again. Not only will **gnatprove** tell us what is wrong, it will suggest a solution as well.

Without the precondition the check it provides would have to be implemented as defensive code in the body. One or the other is critical here, but note that we should never need both.

66.7 Proving Abstract Properties

The postcondition on Increment expresses what is, in fact, a unit-level requirement. Successfully proving such requirements is another significant robustness and cost benefit. Together with the proofs for initialization and AoRTE, these proofs ensure program integrity, that is, the program executes within safe boundaries: the control flow of the program is correctly programmed and cannot be circumvented through run-time errors, and data cannot be corrupted.

We can go even further. We can use contracts to express arbitrary abstract properties when such exist. Safety and security properties, for instance, could be expressed as postconditions and then proven by **gnatprove**.

For example, imagine we have a procedure to move a train to a new position on the track, and we want to do so safely, without leading to a collision with another train. Procedure Move, therefore, takes two inputs: a train identifier specifying which train to move, and the intended new position. The procedure's output is a value indicating a motion command to be given to the train in order to go to that new position. If the train cannot go to that new position safely the output command is to stop the train. Otherwise the command is for the train to continue at an indicated speed:

```
type Move_Result is (Full_Speed, Slow_Down, Keep_Going, Stop);

procedure Move
  (Train      : in Train_Id;
   New_Position : in Train_Position;
   Result      : out Move_Result)
with
  Pre  => Valid_Id (Train) and
           Valid_Move (Trains (Train), New_Position) and
           At_Most_One_Train_Per_Track and
           Safe_Signaling,
  Post => At_Most_One_Train_Per_Track and
           Safe_Signaling;

function At_Most_One_Train_Per_Track return Boolean;
function Safe_Signaling return Boolean;
```

The preconditions specify that, given a safe initial state and a valid move, the result of the call will also be a safe state: there will be at most one train per track section and the track signaling system will not allow any unsafe movements.

66.8 Final Comments

Make sure you understand that **gnatprove** does not attempt to prove the program correct as a whole. It attempts to prove language-defined and user-defined assertions about parts of the program, especially individual routines and calls to those routines. Furthermore, **gnatprove** proves the routines correct only to the extent that the user-defined assertions correctly and sufficiently describe and constrain the implementation of the corresponding routines.

Although we are not proving whole program correctness, as you will have seen — and done — we can prove properties than make our software far more robust and bug-free than is possible otherwise. But in addition, consider what proving the unit-level requirements for your procedures and functions would do for the cost of unit testing and system integration. The tests would pass the first time.

However, within the scope of what SPARK can do, not everything can be proven. In some cases that is because the software behavior is not amenable to expression as boolean conditions (for example, a mouse driver). In other cases the source code is beyond the capabilities of the analyzers that actually do the mathematical proof. In these cases the combination of proof and actual test is appropriate, and still less expensive than testing alone.

There is, of course, much more to be said about what can be done with SPARK and **gnatprove**. Those topics are reserved for the *Introduction to SPARK* (page 261) course.

C TO ADA TRANSLATION PATTERNS

67.1 Naming conventions and casing considerations

One question that may arise relatively soon when converting from C to Ada is the style of source code presentation. The Ada language doesn't impose any particular style and for many reasons, it may seem attractive to keep a C-like style — for example, camel casing — to the Ada program.

However, the code in the Ada language standard, most third-party code, and the libraries provided by GNAT follow a specific style for identifiers and reserved words. Using a different style for the rest of the program leads to inconsistencies, thereby decreasing readability and confusing automatic style checkers. For those reasons, it's usually advisable to adopt the Ada style — in which each identifier starts with an upper case letter, followed by lower case letters (or digits), with an underscore separating two "distinct" words within the identifier. Acronyms within identifiers are in upper case. For example, there is a language-defined package named `Ada.Text_Io`. Reserved words are all lower case.

Following this scheme doesn't preclude adding additional, project-specific rules.

67.2 Manually interfacing C and Ada

Before even considering translating code from C to Ada, it's worthwhile to evaluate the possibility of keeping a portion of the C code intact, and only translating selected modules to Ada. This is a necessary evil when introducing Ada to an existing large C codebase, where re-writing the entire code upfront is not practical nor cost-effective.

Fortunately, Ada has a dedicated set of features for interfacing with other languages. The `Interfaces` package hierarchy and the pragmas `Convention`, `Import`, and `Export` allow you to make inter-language calls while observing proper data representation for each language.

Let's start with the following C code:

[C]

Listing 1: call.c

```
1 #include <stdio.h>
2
3 struct my_struct {
4     int A, B;
5 };
6
7 void call (struct my_struct *p) {
8     printf ("%d", p->A);
9 }
```

To call that function from Ada, the Ada compiler requires a description of the data structure to pass as well as a description of the function itself. To capture how the C **struct my_struct** is represented, we can use the following record along with a **pragma Convention**. The pragma directs the compiler to lay out the data in memory the way a C compiler would.

[Ada]

Listing 2: use_my_struct.adb

```
1 with Ada.Text_IO;  use Ada.Text_IO;
2 with Interfaces.C;
3
4 procedure Use_My_Struct is
5
6     type my_struct is record
7         A : Interfaces.C.int;
8         B : Interfaces.C.int;
9     end record;
10    pragma Convention (C, my_struct);
11
12    V : my_struct := (A => 1, B => 2);
13 begin
14     Put_Line ("V = (" & Interfaces.C.int'Image (V.A) & Interfaces.C.int'Image (V.B) & ")");
15 end Use_My_Struct;
```

Build output

```
use_my_struct.adb:12:04: warning: "V" is not modified, could be declared constant
→ [-gnatwk]
```

Runtime output

```
V = ( 1 2)
```

Describing a foreign subprogram call to Ada code is called *binding* and it is performed in two stages. First, an Ada subprogram specification equivalent to the C function is coded. A C function returning a value maps to an Ada function, and a void function maps to an Ada procedure. Then, rather than implementing the subprogram using Ada code, we use a **pragma Import**:

```
procedure Call (V : my_struct);
pragma Import (C, Call, "call"); -- Third argument optional
```

The Import pragma specifies that whenever Call is invoked by Ada code, it should invoke the Call function with the C calling convention.

And that's all that's necessary. Here's an example of a call to Call:

[Ada]

Listing 3: use_my_struct.adb

```
1 with Interfaces.C;
2
3 procedure Use_My_Struct is
4
5     type my_struct is record
6         A : Interfaces.C.int;
7         B : Interfaces.C.int;
```

(continues on next page)

(continued from previous page)

```

8   end record;
9   pragma Convention (C, my_struct);
10
11  procedure Call (V : my_struct);
12  pragma Import (C, Call, "call"); -- Third argument optional
13
14  V : my_struct := (A => 1, B => 2);
15 begin
16   Call (V);
17 end Use_My_Struct;

```

67.3 Building and Debugging mixed language code

The easiest way to build an application using mixed C / Ada code is to create a simple project file for **gprbuild** and specify C as an additional language. By default, when using **gprbuild** we only compile Ada source files. To compile C code files as well, we use the Languages attribute and specify c as an option, as in the following example of a project file named *default.gpr*:

```

project Default is

  for Languages use ("ada", "c");
  for Main use ("main.adb");

end Default;

```

Then, we use this project file to build the application by simply calling **gprbuild**. Alternatively, we can specify the project file on the command-line with the -P option — for example, **gprbuild -P default.gpr**. In both cases, **gprbuild** compiles all C source-code file found in the directory and links the corresponding object files to build the executable.

In order to include debug information, you can use **gprbuild -cargs -g**. This option adds debug information based on both C and Ada code to the executable. Alternatively, you can specify a Builder package in the project file and include global compilation switches for each language using the *Global_Compilation_Switches* attribute. For example:

```

project Default is

  for Languages use ("ada", "c");
  for Main use ("main.adb");

  package Builder is
    for Global_Compilation_Switches ("Ada") use ("-g");
    for Global_Compilation_Switches ("C") use ("-g");
  end Builder;

end Default;

```

In this case, you can simply run **gprbuild -P default.gpr** to build the executable.

To debug the executable, you can use programs such as **gdb** or **ddd**, which are suitable for debugging both C and Ada source-code. If you prefer a complete IDE, you may want to look into **GNAT Studio**, which supports building and debugging an application within a single environment, and remotely running applications loaded to various embedded devices. You can find more information about **gprbuild** and **GNAT Studio** in the *Introduction to GNAT Toolchain* (page 935) course.

67.4 Automatic interfacing

It may be useful to start interfacing Ada and C by using automatic binding generators. These can be done either by invoking `gcc -fdump-ada-spec` option (to generate an Ada binding to a C header file) or `-gnatceg` option (to generate a C binding to an Ada specification file). For example:

```
gcc -c -fdump-ada-spec my_header.h
gcc -c -gnatceg spec.ads
```

The level of interfacing is very low level and typically requires either massaging (changing the generated files) or wrapping (calling the generated files from a higher level interface). For example, numbers bound from C to Ada are only standard numbers where user-defined types may be desirable. C uses a lot of by-pointer parameters which may be better replaced by other parameter modes, etc.

However, the automatic binding generator helps having a starting point which ensures compatibility of the Ada and the C code.

67.5 Using Arrays in C interfaces

It is relatively straightforward to pass an array from Ada to C. In particular, with the GNAT compiler, passing an array is equivalent to passing a pointer to its first element. Of course, as there's no notion of boundaries in C, the length of the array needs to be passed explicitly. For example:

[C]

Listing 4: p.h

```
1 void p (int * a, int length);
```

[Ada]

Listing 5: main.adb

```
1 procedure Main is
2     type Arr is array (Integer range <>) of Integer;
3
4     procedure P (V : Arr; Length : Integer);
5     pragma Import (C, P);
6
7     X : Arr (5 .. 15);
8 begin
9     P (X, X'Length);
10 end Main;
```

The other way around — that is, retrieving an array that has been creating on the C side — is more difficult. Because C doesn't explicitly carry boundaries, they need to be recreated in some way.

The first option is to actually create an Ada array without boundaries. This is the most flexible, but also the least safe option. It involves creating an array with indices over the full range of `Integer` without ever creating it from Ada, but instead retrieving it as an access from C. For example:

[C]

Listing 6: f.h

```
1 int * f ();
```

[Ada]

Listing 7: main.adb

```
1 procedure Main is
2   type Arr is array (Integer) of Integer;
3   type Arr_A is access all Arr;
4
5   function F return Arr_A;
6   pragma Import (C, F);
7 begin
8   null;
9 end Main;
```

Note that Arr is a constrained type (it doesn't have the `range` \leftrightarrow notation for indices). For that reason, as it would be for C, it's possible to iterate over the whole range of integer, beyond the memory actually allocated for the array.

A somewhat safer way is to overlay an Ada array over the C one. This requires having access to the length of the array. This time, let's consider two cases, one with an array and its size accessible through functions, another one on global variables. This time, as we're using an overlay, the function will be directly mapped to an Ada function returning an address:

[C]

Listing 8: fg.h

```
1 int * f_arr (void);
2 int f_size (void);
3
4 int * g_arr;
5 int g_size;
```

[Ada]

Listing 9: fg.ads

```
1 with System;
2
3 package Fg is
4
5   type Arr is array (Integer range <>) of Integer;
6
7   function F_Arr return System.Address;
8   pragma Import (C, F_Arr, "f_arr");
9
10  function F_Size return Integer;
11  pragma Import (C, F_Size, "f_size");
12
13  F : Arr (0 .. F_Size - 1) with Address => F_Arr;
14
15  G_Size : Integer;
16  pragma Import (C, G_Size, "g_size");
17
18  G_Arr : Arr (0 .. G_Size - 1);
19  pragma Import (C, G_Arr, "g_arr");
```

(continues on next page)

```
21 end Fg;
```

Listing 10: main.adb

```
1 with Fg;
2
3 procedure Main is
4 begin
5   null;
6 end Main;
```

With all solutions though, importing an array from C is a relatively unsafe pattern, as there's only so much information on the array as there would be on the C side in the first place. These are good places for careful peer reviews.

67.6 By-value vs. by-reference types

When interfacing Ada and C, the rules of parameter passing are a bit different with regards to what's a reference and what's a copy. Scalar types and pointers are passed by value, whereas record and arrays are (almost) always passed by reference. However, there may be cases where the C interface also passes values and not pointers to objects. Here's a slightly modified version of a previous example to illustrate this point:

[C]

Listing 11: call.c

```
1 #include <stdio.h>
2
3 struct my_struct {
4   int A, B;
5 };
6
7 void call (struct my_struct p) {
8   printf ("%d", p.A);
9 }
```

In Ada, a type can be modified so that parameters of this type can always be passed by copy.

[Ada]

Listing 12: main.adb

```
1 with Interfaces.C;
2
3 procedure Main is
4   type my_struct is record
5     A : Interfaces.C.int;
6     B : Interfaces.C.int;
7   end record
8   with Convention => C_Pass_By_Copy;
9
10  procedure Call (V : my_struct);
11  pragma Import (C, Call, "call");
12 begin
13   null;
14 end Main;
```

Note that this cannot be done at the subprogram declaration level, so if there is a mix of by-copy and by-reference calls, two different types need to be used on the Ada side.

67.7 Naming and prefixes

Because of the absence of namespaces, any global name in C tends to be very long. And because of the absence of overloading, they can even encode type names in their type.

In Ada, the package is a namespace — two entities declared in two different packages are clearly identified and can always be specifically designated. The C names are usually a good indication of the names of the future packages and should be stripped — it is possible to use the full name if useful. For example, here's how the following declaration and call could be translated:

[C]

Listing 13: reg_interface.h

```
1 void registerInterface_Initialize (int size);
```

Listing 14: reg_interface_test.c

```
1 #include "reg_interface.h"
2
3 int main(int argc, const char * argv[])
4 {
5     registerInterface_Initialize(15);
6
7     return 0;
8 }
```

[Ada]

Listing 15: register_interface.ads

```
1 package Register_Interface is
2     procedure Initialize (Size : Integer)
3         with Import    => True,
4              Convention => C,
5              External_Name => "registerInterface_Initialize";
6
7 end Register_Interface;
```

Listing 16: main.adb

```
1 with Register_Interface;
2
3 procedure Main is
4 begin
5     Register_Interface.Initialize (15);
6 end Main;
```

Note that in the above example, a `use` clause on `Register_Interface` could allow us to omit the prefix.

67.8 Pointers

The first thing to ask when translating pointers from C to Ada is: are they needed in the first place? In Ada, pointers (or access types) should only be used with complex structures that cannot be allocated at run-time — think of a linked list or a graph for example. There are many other situations that would need a pointer in C, but do not in Ada, in particular:

- Arrays, even when dynamically allocated
- Results of functions
- Passing large structures as parameters
- Access to registers
- ... others

This is not to say that pointers aren't used in these cases but, more often than not, the pointer is hidden from the user and automatically handled by the code generated by the compiler; thus avoiding possible mistakes from being made. Generally speaking, when looking at C code, it's good practice to start by analyzing how many pointers are used and to translate as many as possible into *pointerless* Ada structures.

Here are a few examples of such patterns — additional examples can be found throughout this document.

Dynamically allocated arrays can be directly allocated on the stack:

[C]

Listing 17: array_decl.c

```
1 #include <stdlib.h>
2
3 int main() {
4     int *a = malloc(sizeof(int) * 10);
5
6     return 0;
7 }
```

[Ada]

Listing 18: main.adb

```
1 procedure Main is
2     type Arr is array (Integer range <>) of Integer;
3     A : Arr (0 .. 9);
4 begin
```

(continues on next page)

(continued from previous page)

```

5   null;
6 end Main;
```

Build output

```
main.adb:3:04: warning: variable "A" is never read and never assigned [-gnatwv]
```

It's even possible to create a such an array within a structure, provided that the size of the array is known when instantiating this object, using a type discriminant:

[C]

Listing 19: array_decl.c

```

1 #include <stdlib.h>
2
3 typedef struct {
4     int * a;
5 } S;
6
7 int main(int argc, const char * argv[])
8 {
9     S v;
10
11     v.a = malloc(sizeof(int) * 10);
12
13     return 0;
14 }
```

[Ada]

Listing 20: main.adb

```

1 procedure Main is
2     type Arr is array (Integer range <>) of Integer;
3
4     type S (Last : Integer) is record
5         A : Arr (0 .. Last);
6     end record;
7
8     V : S (9);
9 begin
10     null;
11 end Main;
```

Build output

```
main.adb:8:04: warning: variable "V" is never read and never assigned [-gnatwv]
```

With regards to parameter passing, usage mode (input / output) should be preferred to implementation mode (by copy or by reference). The Ada compiler will automatically pass a reference when needed. This works also for smaller objects, so that the compiler will copy in an out when needed. One of the advantages of this approach is that it clarifies the nature of the object: in particular, it differentiates between arrays and scalars. For example:

[C]

Listing 21: p.h

```

1 void p (int * a, int * b);
```

[Ada]

Listing 22: array_types.ads

```

1 package Array_Types is
2     type Arr is array (Integer range <>) of Integer;
3
4     procedure P (A : in out Integer; B : in out Arr);
5 end Array_Types;
```

Most of the time, access to registers end up in some specific structures being mapped onto a specific location in memory. In Ada, this can be achieved through an **Address** clause associated to a variable, for example:

[C]

Listing 23: test_c.c

```

1 int main(int argc, const char * argv[])
2 {
3     int * r = (int *)0xFFFF00A0;
4
5     return 0;
6 }
```

[Ada]

Listing 24: test.adb

```

1 with System;
2
3 procedure Test is
4     R : Integer with Address => System'To_Address (16#FFFF00A0#);
5 begin
6     null;
7 end Test;
```

These are some of the most common misuse of pointers in Ada. Previous sections of the document deal with specifically using access types if absolutely necessary.

67.9 Bitwise Operations

Bitwise operations such as masks and shifts in Ada should be relatively rarely needed, and, when translating C code, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other.

Consider the case of using a C primitive type as a container for single bit boolean flags. In C, this would be done through masks, e.g.:

[C]

Listing 25: flags.c

```

1 #define FLAG_1 0b0001
2 #define FLAG_2 0b0010
3 #define FLAG_3 0b0100
4 #define FLAG_4 0b1000
5
6 int main(int argc, const char * argv[])
7 }
```

(continues on next page)

(continued from previous page)

```

7   int value = 0;
8
9   value |= FLAG_2 | FLAG_4;
10
11  return 0;
12
13 }
```

In Ada, the above can be represented through a Boolean array of enumerate values:

[Ada]

Listing 26: main.adb

```

1 procedure Main is
2   type Values is (Flag_1, Flag_2, Flag_3, Flag_4);
3   type Value_Array is array (Values) of Boolean
4     with Pack;
5
6   Value : Value_Array :=
7     (Flag_2 => True,
8      Flag_4 => True,
9      others => False);
10 begin
11   null;
12 end Main;
```

Build output

```

main.adb:2:20: warning: literal "Flag_1" is not referenced [-gnatwu]
main.adb:2:36: warning: literal "Flag_3" is not referenced [-gnatwu]
main.adb:6:04: warning: variable "Value" is not referenced [-gnatwu]
```

Note the Pack directive for the array, which requests that the array takes as little space as possible.

It is also possible to map records on memory when additional control over the representation is needed or more complex data are used:

[C]

Listing 27: struct_map.c

```

1 int main(int argc, const char * argv[])
2 {
3   int value = 0;
4
5   value = (2 << 1) | 1;
6
7   return 0;
8 }
```

[Ada]

Listing 28: main.adb

```

1 procedure Main is
2   type Value_Rec is record
3     V1 : Boolean;
4     V2 : Integer range 0 .. 3;
5   end record;
```

(continues on next page)

(continued from previous page)

```

7   for Value_Rec use record
8     V1 at 0 range 0 .. 0;
9     V2 at 0 range 1 .. 2;
10    end record;
11
12  Value : Value_Rec := (V1 => True, V2 => 2);
13 begin
14   null;
15 end Main;
```

Build output

```
main.adb:12:04: warning: variable "Value" is not referenced [-gnatwu]
```

The benefit of using Ada structure instead of bitwise operations is threefold:

- The code is simpler to read / write and less error-prone
- Individual fields are named
- The compiler can run consistency checks (for example, check that the value indeed fit in the expected size).

Note that, in cases where bitwise operators are needed, Ada provides modular types with **and**, **or** and **xor** operators. Further shift operators can also be provided upon request through a **pragma**. So the above could also be literally translated to:

[Ada]

Listing 29: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Main is
4    type Value_Type is mod 2 ** 32;
5    pragma Provide_Shift_Operators (Value_Type);
6
7    Value : Value_Type;
8  begin
9    Value := Shift_Left (2, 1) or 1;
10   Put_Line ("Value = " & Value_Type'Image (Value));
11 end Main;
```

Runtime output

```
Value = 5
```

67.10 Mapping Structures to Bit-Fields

In the previous section, we've seen how to perform bitwise operations. In this section, we look at how to interpret a data type as a bit-field and perform low-level operations on it.

In general, you can create a bit-field from any arbitrary data type. First, we declare a bit-field type like this:

[Ada]

```
type Bit_Field is array (Natural range <>) of Boolean with Pack;
```

As we've seen previously, the Pack aspect declared at the end of the type declaration indicates that the compiler should optimize for size. We must use this aspect to be able to interpret data types as a bit-field.

Then, we can use the Size and the **Address** attributes of an object of any type to declare a bit-field for this object. We've discussed the Size attribute *earlier in this course* (page 714).

The **Address** attribute indicates the address in memory of that object. For example, assuming we've declare a variable V, we can declare an actual bit-field object by referring to the **Address** attribute of V and using it in the declaration of the bit-field, as shown here:

[Ada]

```
B : Bit_Field (0 .. V'Size - 1) with Address => V'Address;
```

Note that, in this declaration, we're using the **Address** attribute of V for the **Address** aspect of B.

This technique is called overlays for serialization. Now, any operation that we perform on B will have a direct impact on V, since both are using the same memory location.

The approach that we use in this section relies on the **Address** aspect. Another approach would be to use unchecked conversions, which we'll discuss in the *next section* (page 767).

We should add the Volatile aspect to the declaration to cover the case when both objects can still be changed independently — they need to be volatile, otherwise one change might be missed. This is the updated declaration:

[Ada]

```
B : Bit_Field (0 .. V'Size - 1) with Address => V'Address, Volatile;
```

Using the **Volatile** aspect is important at high level of optimizations. You can find further details about this aspect in the section about the *Volatile and Atomic aspects* (page 710).

Another important aspect that should be added is Import. When used in the context of object declarations, it'll avoid default initialization which could overwrite the existing content while creating the overlay — see an example in the admonition below. The declaration now becomes:

```
B : Bit_Field (0 .. V'Size - 1)
  with
    Address => V'Address, Import, Volatile;
```

Let's look at a simple example:

[Ada]

Listing 30: simple_bitfield.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Simple_Bitfield is
4    type Bit_Field is array (Natural range <>) of Boolean with Pack;
5
6    V : Integer := 0;
7    B : Bit_Field (0 .. V'Size - 1)
8      with Address => V'Address, Import, Volatile;
9  begin
10    B (2) := True;
11    Put_Line ("V = " & Integer'Image (V));
12  end Simple_Bitfield;
```

Runtime output

```
V = 4
```

In this example, we first initialize V with zero. Then, we use the bit-field B and set the third element (B (2)) to **True**. This automatically sets bit #3 of V to 1. Therefore, as expected, the application displays the message V = 4, which corresponds to $2^2 = 4$.

Note that, in the declaration of the bit-field type above, we could also have used a positive range. For example:

```
type Bit_Field is array (Positive range <>) of Boolean with Pack;  
  
B : Bit_Field (1 .. V'Size)  
  with Address => V'Address, Import, Volatile;
```

The only difference in this case is that the first bit is B (1) instead of B (0).

In C, we would rely on bit-shifting and masking to set that specific bit:

[C]

Listing 31: bitfield.c

```
1 #include <stdio.h>  
2  
3 int main(int argc, const char * argv[])
4 {
5     int v = 0;
6
7     v = v | (1 << 2);
8
9     printf("v = %d\n", v);
10
11    return 0;
12 }
```

Runtime output

```
v = 4
```

Important

Ada has the concept of default initialization. For example, you may set the default value of record components:

[Ada]

Listing 32: main.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Main is
4
5     type Rec is record
6         X : Integer := 10;
7         Y : Integer := 11;
8     end record;
9
10    R : Rec;
11 begin
12     Put_Line ("R.X = " & Integer'Image (R.X));
13     Put_Line ("R.Y = " & Integer'Image (R.Y));
14 end Main;
```

Runtime output

```
R.X = 10
R.Y = 11
```

In the code above, we don't explicitly initialize the components of R, so they still have the default values 10 and 11, which are displayed by the application.

Likewise, the `Default_Value` aspect can be used to specify the default value in other kinds of type declarations. For example:

[Ada]

Listing 33: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5    type Percentage is range 0 .. 100
6    with Default_Value => 10;
7
8    P : Percentage;
9  begin
10   Put_Line ("P = " & Percentage'Image (P));
11 end Main;
```

Runtime output

```
P = 10
```

When declaring an object whose type has a default value, the object will automatically be initialized with the default value. In the example above, P is automatically initialized with 10, which is the default value of the Percentage type.

Some types have an implicit default value. For example, access types have a default value of `null`.

As we've just seen, when declaring objects for types with associated default values, automatic initialization will happen. This can also happen when creating an overlay with the `Address` aspect. The default value is then used to overwrite the content at the memory location indicated by the address. However, in most situations, this isn't the behavior we expect, since overlays are usually created to analyze and manipulate existing values. Let's look at an example where this happens:

[Ada]

Listing 34: p.ads

```
1  package P is
2
3    type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;
4
5    type Byte_Field is array (Natural range <>) of Unsigned_8;
6
7    procedure Display_Bytes_Increment (V : in out Integer);
8  end P;
```

Listing 35: p.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body P is
```

(continues on next page)

(continued from previous page)

```

4   procedure Display_Bytes_Increment (V : in out Integer) is
5     BF : Byte_Field (1 .. V'Size / 8)
6       with Address => V'Address, Volatile;
7 begin
8   for B of BF loop
9     Put_Line ("Byte = " & Unsigned_8'Image (B));
10  end loop;
11  Put_Line ("Now incrementing... ");
12  V := V + 1;
13  end Display_Bytes_Increment;
14
15 end P;
16

```

Listing 36: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with P; use P;
4
5  procedure Main is
6    V : Integer := 10;
7 begin
8   Put_Line ("V = " & Integer'Image (V));
9   Display_Bytes_Increment (V);
10  Put_Line ("V = " & Integer'Image (V));
11 end Main;

```

Build output

```

p.adb:7:14: warning: default initialization of "Bf" may modify "V" [enabled by default]
p.adb:7:14: warning: use pragma Import for "Bf" to suppress initialization (RM B. 1(24)) [enabled by default]

```

Runtime output

```

V = 10
Byte = 0
Byte = 0
Byte = 0
Byte = 0
Now incrementing...
V = 1

```

In this example, we expect `Display_Bytes_Increment` to display each byte of the `V` parameter and then increment it by one. Initially, `V` is set to 10, and the call to `Display_Bytes_Increment` should change it to 11. However, due to the default value associated to the `Unsigned_8` type — which is set to 0 — the value of `V` is overwritten in the declaration of `BF` (in `Display_Bytes_Increment`). Therefore, the value of `V` is 1 after the call to `Display_Bytes_Increment`. Of course, this is not the behavior that we originally intended.

Using the `Import` aspect solves this problem. This aspect tells the compiler to not apply default initialization in the declaration because the object is imported. Let's look at the corrected example:

[Ada]

Listing 37: p.ads

```

1 package P is
2
3   type Unsigned_8 is mod 2 ** 8 with Default_Value => 0;
4
5   type Byte_Field is array (Natural range <>) of Unsigned_8;
6
7   procedure Display_Bytes_Increment (V : in out Integer);
8 end P;

```

Listing 38: p.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5   procedure Display_Bytes_Increment (V : in out Integer) is
6     BF : Byte_Field (1 .. V'Size / 8)
7       with Address => V'Address, Import, Volatile;
8 begin
9   for B of BF loop
10    Put_Line ("Byte = " & Unsigned_8'Image (B));
11   end loop;
12   Put_Line ("Now incrementing... ");
13   V := V + 1;
14 end Display_Bytes_Increment;
15
16 end P;

```

Listing 39: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with P; use P;
4
5 procedure Main is
6   V : Integer := 10;
7 begin
8   Put_Line ("V = " & Integer'Image (V));
9   Display_Bytes_Increment (V);
10  Put_Line ("V = " & Integer'Image (V));
11 end Main;

```

Runtime output

```

V = 10
Byte = 10
Byte = 0
Byte = 0
Byte = 0
Now incrementing...
V = 11

```

This unwanted side-effect of the initialization by the `Default_Value` aspect that we've just seen can also happen in these cases:

- when we set a default value for components of a record type declaration,
- when we use the `Default_Component_Value` aspect for array types, or
- when we set use the `Initialize_Scalars` pragma for a package.

Again, using the Import aspect when declaring the overlay eliminates this side-effect.

We can use this pattern for objects of more complex data types like arrays or records. For example:

[Ada]

Listing 40: int_array_bitfield.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Int_Array_Bitfield is
4     type Bit_Field is array (Natural range <>) of Boolean with Pack;
5
6     A : array (1 .. 2) of Integer := (others => 0);
7     B : Bit_Field (0 .. A'Size - 1)
8         with Address => A'Address, Import, Volatile;
9 begin
10    B (2) := True;
11    for I in A'Range loop
12        Put_Line ("A (" & Integer'Image (I)
13                      & ") = " & Integer'Image (A (I)));
14    end loop;
15 end Int_Array_Bitfield;
```

Runtime output

```
A ( 1)= 4
A ( 2)= 0
```

In the Ada example above, we're using the bit-field to set bit #3 of the first element of the array (A (1)). We could set bit #4 of the second element by using the size of the data type (in this case, `Integer'Size`):

[Ada]

```
B (Integer'Size + 3) := True;
```

In C, we would select the specific array position and, again, rely on bit-shifting and masking to set that specific bit:

[C]

Listing 41: bitfield_int_array.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int i;
6     int a[2] = {0, 0};
7
8     a[0] = a[0] | (1 << 2);
9
10    for (i = 0; i < 2; i++)
11    {
12        printf("a[%d] = %d\n", i, a[i]);
13    }
14
15    return 0;
16}
```

Runtime output

```
a[0] = 4
a[1] = 0
```

Since we can use this pattern for any arbitrary data type, this allows us to easily create a subprogram to serialize data types and, for example, transmit complex data structures as a bitstream. For example:

[Ada]

Listing 42: serializer.ads

```
1 package Serializer is
2
3     type Bit_Field is array (Natural range <>) of Boolean with Pack;
4
5     procedure Transmit (B : Bit_Field);
6
7 end Serializer;
```

Listing 43: serializer.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Serializer is
4
5     procedure Transmit (B : Bit_Field) is
6
7         procedure Show_Bit (V : Boolean) is
8             begin
9                 case V is
10                     when False => Put ("0");
11                     when True   => Put ("1");
12                 end case;
13             end Show_Bit;
14
15     begin
16         Put ("Bits: ");
17         for E of B loop
18             Show_Bit (E);
19         end loop;
20         New_Line;
21     end Transmit;
22
23 end Serializer;
```

Listing 44: my_recs.ads

```
1 package My_Recs is
2
3     type Rec is record
4         V : Integer;
5         S : String (1 .. 3);
6     end record;
7
8 end My_Recs;
```

Listing 45: main.adb

```
1 with Serializer;  use Serializer;
2 with My_Recs;    use My_Recs;
```

(continues on next page)

(continued from previous page)

```

4  procedure Main is
5    R : Rec := (5, "abc");
6    B : Bit_Field (0 .. R'Size - 1)
7    with Address => R'Address, Import, Volatile;
8  begin
9    Transmit (B);
10 end Main;

```

Build output

```
main.adb:9:14: warning: volatile actual passed by copy (RM C.6(19)) [enabled by default]
```

Runtime output

```
Bits: 1010000000000000000000000000000010000110010001101100011000000000
```

In this example, the `Transmit` procedure from `Serializer` package displays the individual bits of a bit-field. We could have used this strategy to actually transmit the information as a bitstream. In the main application, we call `Transmit` for the object `R` of record type `Rec`. Since `Transmit` has the bit-field type as a parameter, we can use it for any type, as long as we have a corresponding bit-field representation.

In C, we interpret the input pointer as an array of bytes, and then use shifting and masking to access the bits of that byte. Here, we use the `char` type because it has a size of one byte in most platforms.

[C]

Listing 46: my_recs.h

```

1  typedef struct {
2    int v;
3    char s[4];
4 } rec;

```

Listing 47: serializer.h

```
1  void transmit (void *bits, int len);
```

Listing 48: serializer.c

```

1  #include "serializer.h"
2
3  #include <stdio.h>
4  #include <assert.h>
5
6  void transmit (void *bits, int len)
7  {
8    int i, j;
9    char *c = (char *)bits;
10
11   assert(sizeof(char) == 1);
12
13   printf("Bits: ");
14   for (i = 0; i < len / (sizeof(char) * 8); i++)
15   {
16     for (j = 0; j < sizeof(char) * 8; j++)
17     {
18       printf("%d", c[i] >> j & 1);

```

(continues on next page)

(continued from previous page)

```

19         }
20     }
21     printf("\n");
22 }
```

Listing 49: bitfield_serialization.c

```

1 #include <stdio.h>
2
3 #include "my_recs.h"
4 #include "serializer.h"
5
6 int main(int argc, const char * argv[])
7 {
8     rec r = {5, "abc"};
9
10    transmit(&r, sizeof(r) * 8);
11
12    return 0;
13 }
```

Runtime output

```
Bits: 1010000000000000000000000000000010000110010001101100011000000000
```

Similarly, we can write a subprogram that converts a bit-field — which may have been received as a bitstream — to a specific type. We can add a To_Rec subprogram to the My_Recs package to convert a bit-field to the Rec type. This can be used to convert a bitstream that we received into the actual data type representation.

As you know, we may write the To_Rec subprogram as a procedure or as a function. Since we need to use slightly different strategies for the implementation, the following example has both versions of To_Rec.

This is the updated code for the My_Recs package and the Main procedure:

[Ada]

Listing 50: serializer.ads

```

1 package Serializer is
2
3     type Bit_Field is array (Natural range <>) of Boolean with Pack;
4
5     procedure Transmit (B : Bit_Field);
6
7 end Serializer;
```

Listing 51: serializer.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Serializer is
4
5     procedure Transmit (B : Bit_Field) is
6
7         procedure Show_Bit (V : Boolean) is
8             begin
9                 case V is
10                     when False => Put ("0");
11                     when True   => Put ("1");
```

(continues on next page)

(continued from previous page)

```

12      end case;
13  end Show_Bit;

14
15 begin
16   Put ("Bits: ");
17   for E of B loop
18     Show_Bit (E);
19   end loop;
20   New_Line;
21 end Transmit;

22
23 end Serializer;

```

Listing 52: my_recs.ads

```

1 with Serializer;  use Serializer;
2
3 package My_Recs is
4
5   type Rec is record
6     V : Integer;
7     S : String (1 .. 3);
8   end record;
9
10  procedure To_Rec (B : Bit_Field;
11                     R : out Rec);
12
13  function To_Rec (B : Bit_Field) return Rec;
14
15  procedure Display (R : Rec);
16
17 end My_Recs;

```

Listing 53: my_recs.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body My_Recs is
4
5   procedure To_Rec (B : Bit_Field;
6                     R : out Rec) is
7     B_R : Rec
8       with Address => B'Address, Import, Volatile;
9   begin
10    -- Assigning data from overlayed record B_R to output parameter R.
11    R := B_R;
12  end To_Rec;
13
14  function To_Rec (B : Bit_Field) return Rec is
15    R : Rec;
16    B_R : Rec
17      with Address => B'Address, Import, Volatile;
18  begin
19    -- Assigning data from overlayed record B_R to local record R.
20    R := B_R;
21
22    return R;
23  end To_Rec;
24
25  procedure Display (R : Rec) is

```

(continues on next page)

(continued from previous page)

```

26 begin
27   Put ("(" & Integer'Image (R.V) & ", "
28       & (R.S) & ")");
29 end Display;
30
31 end My_Recs;

```

Listing 54: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2  with Serializer;  use Serializer;
3  with My_Recs;    use My_Recs;
4
5  procedure Main is
6    R1 : Rec := (5, "abc");
7    R2 : Rec := (0, "zzz");
8
9    B1 : Bit_Field (0 .. R1'Size - 1)
10   with Address => R1'Address, Import, Volatile;
11 begin
12   Put ("R2 = ");
13   Display (R2);
14   New_Line;
15
16   -- Getting Rec type using data from B1, which is a bit-field
17   -- representation of R1.
18   To_Rec (B1, R2);
19
20   -- We could use the function version of To_Rec:
21   -- R2 := To_Rec (B1);
22
23   Put_Line ("New bitstream received!");
24   Put ("R2 = ");
25   Display (R2);
26   New_Line;
27 end Main;

```

Build output

```
main.adb:18:12: warning: volatile actual passed by copy (RM C.6(19)) [enabled by default]
```

Runtime output

```
R2 = ( 0, zzz)
New bitstream received!
R2 = ( 5, abc)
```

In both versions of `To_Rec`, we declare the record object `B_R` as an overlay of the input bit-field. In the procedure version of `To_Rec`, we then simply copy the data from `B_R` to the output parameter `R`. In the function version of `To_Rec`, however, we need to declare a local record object `R`, which we return after the assignment.

In C, we can interpret the input pointer as an array of bytes, and copy the individual bytes. For example:

[C]

Listing 55: my_recs.h

```

1 typedef struct {
2     int v;
3     char s[3];
4 } rec;
5
6 void to_r (void *bits, int len, rec *r);
7
8 void display_r (rec *r);

```

Listing 56: my_recs.c

```

1 #include "my_recs.h"
2
3 #include <stdio.h>
4 #include <assert.h>
5
6 void to_r (void *bits, int len, rec *r)
7 {
8     int i;
9     char *c1 = (char *)bits;
10    char *c2 = (char *)r;
11
12    assert(len == sizeof(rec) * 8);
13
14    for (i = 0; i < len / (sizeof(char) * 8); i++)
15    {
16        c2[i] = c1[i];
17    }
18}
19
20 void display_r (rec *r)
21 {
22     printf("{d, %c%c%c}", r->v, r->s[0], r->s[1], r->s[2]);
23 }

```

Listing 57: bitfield_serialization.c

```

1 #include <stdio.h>
2 #include "my_recs.h"
3
4 int main(int argc, const char * argv[])
5 {
6     rec r1 = {5, "abc"};
7     rec r2 = {0, "zzz"};
8
9     printf("r2 = ");
10    display_r (&r2);
11    printf("\n");
12
13    to_r(&r1, sizeof(r1) * 8, &r2);
14
15    printf("New bitstream received!\n");
16    printf("r2 = ");
17    display_r (&r2);
18    printf("\n");
19
20    return 0;
21 }

```

Runtime output

```
r2 = {0, zzz}
New bitstream received!
r2 = {5, abc}
```

Here, `to_r` casts both pointer parameters to pointers to `char` to get a byte-aligned pointer. Then, it simply copies the data byte-by-byte.

67.10.1 Overlays vs. Unchecked Conversions

Unchecked conversions are another way of converting between unrelated data types. This conversion is done by instantiating the generic `Unchecked_Conversions` function for the types you want to convert. Let's look at a simple example:

[Ada]

Listing 58: simple_unchecked_conversion.adb

```
1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Unchecked_Conversion;
3
4  procedure Simple_Unchecked_Conversion is
5    type State is (Off, State_1, State_2)
6    with Size => Integer'Size;
7
8    for State use (Off => 0, State_1 => 32, State_2 => 64);
9
10   function As_Integer is new Ada.Unchecked_Conversion (Source => State,
11                                         Target => Integer);
12
13   I : Integer;
14 begin
15   I := As_Integer (State_2);
16   Put_Line ("I = " & Integer'Image (I));
17 end Simple_Unchecked_Conversion;
```

Runtime output

```
I = 64
```

In this example, `As_Integer` is an instantiation of `Unchecked_Conversion` to convert between the `State` enumeration and the `Integer` type. Note that, in order to ensure safe conversion, we're declaring `State` to have the same size as the `Integer` type we want to convert to.

This is the corresponding implementation using overlays:

[Ada]

Listing 59: simple_overlay.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Simple_Overlay is
4    type State is (Off, State_1, State_2)
5    with Size => Integer'Size;
6
7    for State use (Off => 0, State_1 => 32, State_2 => 64);
8
9    S : State;
10   I : Integer
```

(continues on next page)

(continued from previous page)

```

11   with Address => S'Address, Import, Volatile;
12 begin
13   S := State_2;
14   Put_Line ("I = " & Integer'Image (I));
15 end Simple_Overlay;
```

Runtime output

```
I = 64
```

Let's look at another example of converting between different numeric formats. In this case, we want to convert between a 16-bit fixed-point and a 16-bit integer data type. This is how we can do it using `Unchecked_Conversion`:

[Ada]

Listing 60: `fixed_int_unchecked_conversion.adb`

```

1  with Ada.Text_Io;           use Ada.Text_Io;
2  with Ada.Unchecked_Conversion;
3
4  procedure Fixed_Int_Unchecked_Conversion is
5    Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
6    Max_16   : constant := 2 ** 15;
7
8    type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
9      with Size => 16;
10   type Int_16   is range -Max_16 .. Max_16 - 1
11     with Size => 16;
12
13  function As_Int_16 is new Ada.Unchecked_Conversion (Source => Fixed_16,
14                                         Target => Int_16);
15  function As_Fixed_16 is new Ada.Unchecked_Conversion (Source => Int_16,
16                                         Target => Fixed_16);
17
18  I : Int_16 := 0;
19  F : Fixed_16 := 0.0;
20 begin
21  F := Fixed_16'Last;
22  I := As_Int_16 (F);
23
24  Put_Line ("F = " & Fixed_16'Image (F));
25  Put_Line ("I = " & Int_16'Image (I));
26 end Fixed_Int_Unchecked_Conversion;
```

Build output

```
fixed_int_unchecked_conversion.adb:15:13: warning: function "As_Fixed_16" is not
referenced [-gnatwu]
```

Runtime output

```
F = 0.99997
I = 32767
```

Here, we instantiate `Unchecked_Conversion` for the `Int_16` and `Fixed_16` types, and we call the instantiated functions explicitly. In this case, we call `As_Int_16` to get the integer value corresponding to `Fixed_16'Last`.

This is how we can rewrite the implementation above using overlays:

[Ada]

Listing 61: fixed_int_overlay.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Fixed_Int_Overlay is
4      Delta_16 : constant := 1.0 / 2.0 ** (16 - 1);
5      Max_16   : constant := 2 ** 15;
6
7      type Fixed_16 is delta Delta_16 range -1.0 .. 1.0 - Delta_16
8          with Size => 16;
9      type Int_16   is range -Max_16 .. Max_16 - 1
10         with Size => 16;
11
12     I : Int_16   := 0;
13     F : Fixed_16
14         with Address => I'Address, Import, Volatile;
15 begin
16     F := Fixed_16'Last;
17
18     Put_Line ("F = " & Fixed_16'Image (F));
19     Put_Line ("I = " & Int_16'Image (I));
20 end Fixed_Int_Overlay;

```

Runtime output

```
F = 0.99997
I = 32767
```

Here, the conversion to the integer value is implicit, so we don't need to call a conversion function.

Using `Unchecked_Conversion` has the advantage of making it clear that a conversion is happening, since the conversion is written explicitly in the code. With overlays, that conversion is automatic and therefore implicit. In that sense, using an unchecked conversion is a cleaner and safer approach. On the other hand, an unchecked conversion requires a copy, so it's less efficient than overlays, where no copy is performed — because one change in the source object is automatically reflected in the target object (and vice-versa). In the end, the choice between unchecked conversions and overlays depends on the level of performance that you want to achieve.

Also note that an unchecked conversion only has defined behavior when instantiated for constrained types. For example, we shouldn't use this kind of conversion:

```
Ada.Unchecked_Conversion (Source => String,
                           Target => Integer);
```

Although this compiles, the behavior will only be well-defined in those cases when `Source'Size = Target'Size`. Therefore, instead of using an unconstrained type for `Source`, we should use a subtype that matches this expectation:

```
subtype Integer_String is String (1 .. Integer'Size / Character'Size);

function As_Integer is new
    Ada.Unchecked_Conversion (Source => Integer_String,
                              Target => Integer);
```

Similarly, in order to rewrite the examples using bit-fields that we've seen in the previous section, we cannot simply instantiate `Unchecked_Conversion` with the `Target` indicating the *unconstrained* bit-field, such as:

```
Ada.Unchecked_Conversion (Source => Integer,
                           Target => Bit_Field);
```

Instead, we have to declare a subtype for the specific range we're interested in. This is how we can rewrite one of the previous examples:

[Ada]

Listing 62: simple_bitfield_conversion.adb

```
1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Unchecked_Conversion;
3
4  procedure Simple_Bitfield_Conversion is
5    type Bit_Field is array (Natural range <>) of Boolean with Pack;
6
7    V : Integer := 4;
8
9    -- Declaring subtype that takes the size of V into account.
10   --
11   subtype Integer_Bit_Field is Bit_Field (0 .. V'Size - 1);
12
13   -- NOTE: we could also use the Integer type in the declaration:
14   --
15   -- subtype Integer_Bit_Field is Bit_Field (0 .. Integer'Size - 1);
16   --
17
18   -- Using the Integer_Bit_Field subtype as the target
19   function As_Bit_Field is new
20     Ada.Unchecked_Conversion (Source => Integer,
21                               Target => Integer_Bit_Field);
22
23   B : Integer_Bit_Field;
24 begin
25   B := As_Bit_Field (V);
26
27   Put_Line ("V = " & Integer'Image (V));
28 end Simple_Bitfield_Conversion;
```

Build output

```
simple_bitfield_conversion.adb:7:04: warning: "V" is not modified, could be
  ↪declared constant [-gnatwk]
simple_bitfield_conversion.adb:23:04: warning: variable "B" is assigned but never
  ↪read [-gnatwm]
simple_bitfield_conversion.adb:25:04: warning: possibly useless assignment to "B",
  ↪value might not be referenced [-gnatwm]
```

Runtime output

```
V = 4
```

In this example, we first declare the subtype `Integer_Bit_Field` as a bit-field with a length that fits the `V` variable we want to convert to. Then, we can use that subtype in the instantiation of `Unchecked_Conversion`.

HANDLING VARIABILITY AND RE-USABILITY

68.1 Understanding static and dynamic variability

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. For example, the same application may need to be portable between two different architectures (ARM and x86), or two different platforms with different set of devices available. Maybe the same application is used for two different generations of the product, so it needs to account for absence or presence of new features, or it's used for different projects which may select different components or configurations. All these cases, and many others, require variability in the software in order to ensure its reusability.

In C, variability is usually achieved through macros and function pointers, the former being tied to static variability (variability in different builds) the latter to dynamic variability (variability within the same build decided at run-time).

Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. When Ada isn't enough, the GNAT compilation system also provides a layer of capabilities, in particular selection of alternate bodies.

If you're familiar with object-oriented programming (OOP) — supported in languages such as C++ and Java —, you might also be interested in knowing that OOP is supported by Ada and can be used to implement variability. This should, however, be used with care, as OOP brings its own set of problems, such as loss of efficiency — dispatching calls can't be inlined and require one level of indirection — or loss of analyzability — the target of a dispatching call isn't known at run time. As a rule of thumb, OOP should be considered only for cases of dynamic variability, where several versions of the same object need to exist concurrently in the same application.

68.2 Handling variability & reusability statically

68.2.1 Genericity

One usage of C macros involves the creation of functions that works regardless of the type they're being called upon. For example, a swap macro may look like:

[C]

Listing 1: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define SWAP(t, a, b) ({
```

(continues on next page)

(continued from previous page)

```

5           t tmp = a; \
6           a = b; \
7           b = tmp; \
8       })
9
10      int main()
11      {
12          int a = 10;
13          int b = 42;
14
15          printf("a = %d, b = %d\n", a, b);
16
17          SWAP (int, a, b);
18
19          printf("a = %d, b = %d\n", a, b);
20
21          return 0;
22      }

```

Runtime output

```

a = 10, b = 42
a = 42, b = 10

```

Ada offers a way to declare this kind of functions as a generic, that is, a function that is written after static arguments, such as a parameter:

[Ada]

Listing 2: main.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Main is
4
5     generic
6         type A_Type is private;
7         procedure Swap (Left, Right : in out A_Type);
8
9     procedure Swap (Left, Right : in out A_Type) is
10        Temp : constant A_Type := Left;
11    begin
12        Left := Right;
13        Right := Temp;
14    end Swap;
15
16    procedure Swap_I is new Swap (Integer);
17
18    A : Integer := 10;
19    B : Integer := 42;
20
21 begin
22    Put_Line ("A = "
23              & Integer'Image (A)
24              & ", B = "
25              & Integer'Image (B));
26
27    Swap_I (A, B);
28
29    Put_Line ("A = "
30              & Integer'Image (A)

```

(continues on next page)

(continued from previous page)

```

31      & ", B = "
32      & Integer'Image (B));
33 end Main;

```

Runtime output

```
A = 10, B = 42
A = 42, B = 10
```

There are a few key differences between the C and the Ada version here. In C, the macro can be used directly and essentially get expanded by the preprocessor without any kind of checks. In Ada, the generic will first be checked for internal consistency. It then needs to be explicitly instantiated for a concrete type. From there, it's exactly as if there was an actual version of this Swap function, which is going to be called as any other function. All rules for parameter modes and control will apply to this instance.

In many respects, an Ada generic is a way to provide a safe specification and implementation of such macros, through both the validation of the generic itself and its usage.

Subprograms aren't the only entities that can be made generic. As a matter of fact, it's much more common to render an entire package generic. In this case the instantiation creates a new version of all the entities present in the generic, including global variables. For example:

[Ada]

Listing 3: gen.ads

```

1 generic
2   type T is private;
3 package Gen is
4   type C is tagged record
5     V : T;
6   end record;
7
8   G : Integer;
9 end Gen;

```

The above can be instantiated and used the following way:

Listing 4: main.adb

```

1 with Gen;
2
3 procedure Main is
4   package I1 is new Gen (Integer);
5   package I2 is new Gen (Integer);
6   subtype Str10 is String (1 .. 10);
7   package I3 is new Gen (Str10);
8 begin
9   I1.G := 0;
10  I2.G := 1;
11  I3.G := 2;
12 end Main;

```

Here, I1.G, I2.G and I3.G are three distinct variables.

So far, we've only looked at generics with one kind of parameter: a so-called private type. There's actually much more that can be described in this section, such as variables, subprograms or package instantiations with certain properties. For example, the following provides a sort algorithm for any kind of structurally compatible array type:

[Ada]

Listing 5: sort.ads

```

1 generic
2   type Component is private;
3   type Index is (<>);
4   with function "<" (Left, Right : Component) return Boolean;
5   type Array_Type is array (Index range <>) of Component;
6   procedure Sort (A : in out Array_Type);

```

The declaration above states that we need a type (Component), a discrete type (Index), a comparison subprogram ("`<`"), and an array definition (Array_Type). Given these, it's possible to write an algorithm that can sort any Array_Type. Note the usage of the `with` reserved word in front of the function name: it exists to differentiate between the generic parameter and the beginning of the generic subprogram.

Here is a non-exhaustive overview of the kind of constraints that can be put on types:

```

type T is private; -- T is a constrained type, such as Integer
type T (<>) is private; -- T can be an unconstrained type e.g. String
type T is tagged private; -- T is a tagged type
type T is new T2 with private; -- T is an extension of T2
type T is (<>); -- T is a discrete type
type T is range <>; -- T is an integer type
type T is digits <>; -- T is a floating point type
type T is access T2; -- T is an access type to T2

```

For a more complete list please reference the Generic Formal Types in the [Appendix of the Introduction to Ada course](#) (page 255).

68.2.2 Simple derivation

Let's take a case where a codebase needs to handle small variations of a given device, or maybe different generations of a device, depending on the platform it's running on. In this example, we're assuming that each platform will lead to a different binary, so the code can statically resolve which set of services are available. However, we want an easy way to implement a new device based on a previous one, saying "this new device is the same as this previous device, with these new services and these changes in existing services".

We can implement such patterns using Ada's simple derivation — as opposed to tagged derivation, which is OOP-related and discussed in a later section.

Let's start from the following example:

[Ada]

Listing 6: drivers_1.ads

```

1 package Drivers_1 is
2
3   type Device_1 is null record;
4   procedure Startup (Device : Device_1);
5   procedure Send (Device : Device_1; Data : Integer);
6   procedure Send_Fast (Device : Device_1; Data : Integer);
7   procedure Receive (Device : Device_1; Data : out Integer);
8
9 end Drivers_1;

```

Listing 7: drivers_1.adb

```

1 package body Drivers_1 is
2
3     -- NOTE: unimplemented procedures: Startup, Send, Send_Fast
4     --          mock-up implementation: Receive
5
6     procedure Startup (Device : Device_1) is null;
7
8     procedure Send (Device : Device_1; Data : Integer) is null;
9
10    procedure Send_Fast (Device : Device_1; Data : Integer) is null;
11
12    procedure Receive (Device : Device_1; Data : out Integer) is
13        begin
14            Data := 42;
15        end Receive;
16
17 end Drivers_1;

```

In the above example, `Device_1` is an empty record type. It may also have some fields if required, or be a different type such as a scalar. Then the four procedures `Startup`, `Send`, `Send_Fast` and `Receive` are primitives of this type. A primitive is essentially a subprogram that has a parameter or return type directly referencing this type and declared in the same scope. At this stage, there's nothing special with this type: we're using it as we would use any other type. For example:

[Ada]

Listing 8: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers_1;   use Drivers_1;
3
4 procedure Main is
5     D : Device_1;
6     I : Integer;
7 begin
8     Startup (D);
9     Send_Fast (D, 999);
10    Receive (D, I);
11    Put_Line (Integer'Image (I));
12 end Main;

```

Build output

```
drivers_1.adb:12:23: warning: formal parameter "Device" is not referenced [-gnatwu]
```

Runtime output

```
42
```

Let's now assume that we need to implement a new generation of device, `Device_2`. This new device works exactly like the first one, except for the startup code that has to be done differently. We can create a new type that operates exactly like the previous one, but modifies only the behavior of `Startup`:

[Ada]

Listing 9: drivers_2.ads

```
1 with Drivers_1; use Drivers_1;
2
3 package Drivers_2 is
4
5   type Device_2 is new Device_1;
6
7   overriding
8   procedure Startup (Device : Device_2);
9
10 end Drivers_2;
```

Listing 10: drivers_2.adb

```
1 package body Drivers_2 is
2
3   overriding
4   procedure Startup (Device : Device_2) is null;
5
6 end Drivers_2;
```

Here, Device_2 is derived from Device_1. It contains all the exact same properties and primitives, in particular, Startup, Send, Send_Fast and Receive. However, here, we decided to change the Startup function and to provide a different implementation. We override this function. The main subprogram doesn't change much, except for the fact that it now relies on a different type:

[Ada]

Listing 11: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers_2;   use Drivers_2;
3
4 procedure Main is
5   D : Device_2;
6   I : Integer;
7 begin
8   Startup (D);
9   Send_Fast (D, 999);
10  Receive (D, I);
11  Put_Line (Integer'Image (I));
12 end Main;
```

Build output

```
drivers_1.adb:12:23: warning: formal parameter "Device" is not referenced [-gnatwu]
```

Runtime output

```
42
```

We can continue with this approach and introduce a new generation of devices. This new device doesn't implement the Send_Fast service so we want to remove it from the list of available services. Furthermore, for the purpose of our example, let's assume that the hardware team went back to the Device_1 way of implementing Startup. We can write this new device the following way:

[Ada]

Listing 12: drivers_3.ads

```

1  with Drivers_1; use Drivers_1;
2
3  package Drivers_3 is
4
5      type Device_3 is new Device_1;
6
7      overriding
8      procedure Startup (Device : Device_3);
9
10     procedure Send_Fast (Device : Device_3; Data : Integer)
11        is abstract;
12
13 end Drivers_3;

```

Listing 13: drivers_3.adb

```

1  package body Drivers_3 is
2
3      overriding
4      procedure Startup (Device : Device_3) is null;
5
6  end Drivers_3;

```

The **is abstract** definition makes illegal any call to a function, so calls to Send_Fast on Device_3 will be flagged as being illegal. To then implement Startup of Device_3 as being the same as the Startup of Device_1, we can convert the type in the implementation:

[Ada]

Listing 14: drivers_3.adb

```

1  package body Drivers_3 is
2
3      overriding
4      procedure Startup (Device : Device_3) is
5      begin
6          Drivers_1.Startup (Device_1 (Device));
7      end Startup;
8
9  end Drivers_3;

```

Our Main now looks like:

[Ada]

Listing 15: main.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Drivers_3;   use Drivers_3;
3
4  procedure Main is
5      D : Device_3;
6      I : Integer;
7  begin
8      Startup (D);
9      Send_Fast (D, 999);
10     Receive (D, I);
11     Put_Line (Integer'Image (I));
12 end Main;

```

Build output

```
drivers_1.adb:12:23: warning: formal parameter "Device" is not referenced [-gnatwu]
main.adb:9:04: error: cannot call abstract operation "Send_Fast" declared at
  ↪drivers_3.ads:10
gprbuild: *** compilation phase failed
```

Here, the call to `Send_Fast` will get flagged by the compiler.

Note that the fact that the code of Main has to be changed for every implementation isn't necessarily satisfactory. We may want to go one step further, and isolate the selection of the device kind to be used for the whole application in one unique file. One way to do this is to use the same name for all types, and use a renaming to select which package to use. Here's a simplified example to illustrate that:

[Ada]

Listing 16: drivers_1.ads

```
1 package Drivers_1 is
2
3   type Transceiver is null record;
4   procedure Send (Device : Transceiver; Data : Integer);
5   procedure Receive (Device : Transceiver; Data : out Integer);
6
7 end Drivers_1;
```

Listing 17: drivers_1.adb

```
1 package body Drivers_1 is
2
3   procedure Send (Device : Transceiver; Data : Integer) is null;
4
5   procedure Receive (Device : Transceiver; Data : out Integer) is
6     pragma Unreferenced (Device);
7   begin
8     Data := 42;
9   end Receive;
10
11 end Drivers_1;
```

Listing 18: drivers_2.ads

```
1 with Drivers_1;
2
3 package Drivers_2 is
4
5   type Transceiver is new Drivers_1.Transceiver;
6   procedure Send (Device : Transceiver; Data : Integer);
7   procedure Receive (Device : Transceiver; Data : out Integer);
8
9 end Drivers_2;
```

Listing 19: drivers_2.adb

```
1 package body Drivers_2 is
2
3   procedure Send (Device : Transceiver; Data : Integer) is null;
4
5   procedure Receive (Device : Transceiver; Data : out Integer) is
6     pragma Unreferenced (Device);
7   begin
```

(continues on next page)

(continued from previous page)

```

8   Data := 42;
9   end Receive;
10
11 end Drivers_2;
```

Listing 20: drivers.ads

```

1 with Drivers_1;
2
3 package Drivers renames Drivers_1;
```

Listing 21: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Drivers;      use Drivers;
3
4 procedure Main is
5   D : Transceiver;
6   I : Integer;
7 begin
8   Send (D, 999);
9   Receive (D, I);
10  Put_Line (Integer'Image (I));
11 end Main;
```

Runtime output

42

In the above example, the whole code can rely on drivers.ads, instead of relying on the specific driver. Here, Drivers is another name for Driver_1. In order to switch to Driver_2, the project only has to replace that one drivers.ads file.

In the following section, we'll go one step further and demonstrate that this selection can be done through a configuration switch selected at build time instead of a manual code modification.

68.2.3 Configuration pragma files

Configuration pragmas are a set of pragmas that modify the compilation of source-code files. You may use them to either relax or strengthen requirements. For example:

```
pragma Suppress (Overflow_Check);
```

In this example, we're suppressing the overflow check, thereby relaxing a requirement. Normally, the following program would raise a constraint error due to a failed overflow check:

[Ada]

Listing 22: p.ads

```

1 package P is
2   function Add_Max (A : Integer) return Integer;
3 end P;
```

Listing 23: p.adb

```

1 package body P is
2     function Add_Max (A : Integer) return Integer is
3     begin
4         return A + Integer'Last;
5     end Add_Max;
6 end P;

```

Listing 24: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with P;           use P;
3
4 procedure Main is
5     I : Integer := Integer'Last;
6 begin
7     I := Add_Max (I);
8     Put_Line ("I = " & Integer'Image (I));
9 end Main;

```

Runtime output

```
raised CONSTRAINT_ERROR : p.adb:4 overflow check failed
```

When suppressing the overflow check, however, the program doesn't raise an exception, and the value that Add_Max returns is -2, which is a wraparound of the sum of the maximum integer values (`Integer'Last + Integer'Last`).

We could also strengthen requirements, as in this example:

```
pragma Restrictions (No_Floating_Point);
```

Here, the restriction forbids the use of floating-point types and objects. The following program would violate this restriction, so the compiler isn't able to compile the program when the restriction is used:

```

procedure Main is
    F : Float := 0.0;
    -- Declaration is not possible with No_Floating_Point restriction.
begin
    null;
end Main;

```

Restrictions are especially useful for high-integrity applications. In fact, the Ada Reference Manual has a [separate section for them](#)¹⁰⁸.

When creating a project, it is practical to list all configuration pragmas in a separate file. This is called a configuration pragma file, and it usually has an `.adc` file extension. If you use **GPRbuild** for building Ada applications, you can specify the configuration pragma file in the corresponding project file. For example, here we indicate that `gnat.adc` is the configuration pragma file for our project:

```

project Default is

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

```

(continues on next page)

¹⁰⁸ <http://www.adacore.com/standards/12rm/html/RM-H-4.html>

(continued from previous page)

```

package Compiler is
  for Local_Configuration_Pragmas use "gnat.adc";
end Compiler;

end Default;

```

68.2.4 Configuration packages

In C, preprocessing flags are used to create blocks of code that are only compiled under certain circumstances. For example, we could have a block that is only used for debugging:

[C]

Listing 25: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>

3
4 int func(int x)
5 {
6     return x % 4;
7 }

8
9 int main()
10 {
11     int a, b;
12
13     a = 10;
14     b = func(a);
15
16 #ifdef DEBUG
17     printf("func(%d) => %d\n", a, b);
18 #endif
19
20     return 0;
21 }

```

Here, the block indicated by the DEBUG flag is only included in the build if we define this preprocessing flag, which is what we expect for a debug version of the build. In the release version, however, we want to keep debug information out of the build, so we don't use this flag during the build process.

Ada doesn't define a preprocessor as part of the language. Some Ada toolchains — like the GNAT toolchain — do have a preprocessor that could create code similar to the one we've just seen. When programming in Ada, however, the recommendation is to use configuration packages to select code blocks that are meant to be included in the application.

When using a configuration package, the example above can be written as:

[Ada]

Listing 26: config.ads

```

1 package Config is
2
3   Debug : constant Boolean := False;
4
5 end Config;

```

Listing 27: func.ads

```
1 function Func (X : Integer) return Integer;
```

Listing 28: func.adb

```
1 function Func (X : Integer) return Integer is
2 begin
3     return X mod 4;
4 end Func;
```

Listing 29: main.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2 with Config;
3 with Func;
4
5 procedure Main is
6     A, B : Integer;
7 begin
8     A := 10;
9     B := Func (A);
10
11    if Config.Debug then
12        Put_Line ("Func(" & Integer'Image (A) & ") => "
13                  & Integer'Image (B));
14    end if;
15 end Main;
```

In this example, Config is a configuration package. The version of Config we're seeing here is the release version. The debug version of the Config package looks like this:

```
package Config is
    Debug : constant Boolean := True;
end Config;
```

The compiler makes sure to remove dead code. In the case of the release version, since Config.Debug is constant and set to **False**, the compiler is smart enough to remove the call to Put_Line from the build.

As you can see, both versions of Config are very similar to each other. The general idea is to create packages that declare the same constants, but using different values.

In C, we differentiate between the debug and release versions by selecting the appropriate preprocessing flags, but in Ada, we select the appropriate configuration package during the build process. Since the file name is usually the same (config.ads for the example above), we may want to store them in distinct directories. For the example above, we could have:

- src/debug/config.ads for the debug version, and
- src/release/config.ads for the release version.

Then, we simply select the appropriate configuration package for each version of the build by indicating the correct path to it. When using **GPRbuild**, we can select the appropriate directory where the config.ads file is located. We can use scenario variables in our project, which allow for creating different versions of a build. For example:

```
project Default is
```

(continues on next page)

(continued from previous page)

```

type Mode_Type is ("debug", "release");

Mode : Mode_Type := external ("mode", "debug");

for Source_Dirs use ("src", "src/" & Mode);
for Object_Dir use "obj";
for Main use ("main.adb");

end Default;

```

In this example, we're defining a scenario type called `Mode_Type`. Then, we're declaring the scenario variable `Mode` and using it in the `Source_Dirs` declaration to complete the path to the subdirectory containing the `config.ads` file. The expression `"src/" & Mode` concatenates the user-specified mode to select the appropriate subdirectory.

We can then set the mode on the command-line. For example:

```
gprbuild -P default.gpr -Xmode=release
```

In addition to selecting code blocks for the build, we could also specify values that depend on the target build. For our example above, we may want to create two versions of the application, each one having a different version of a `MOD_VALUE` that is used in the implementation of `func()`. In C, we can achieve this by using preprocessing flags and defining the corresponding version in `APP_VERSION`. Then, depending on the value of `APP_VERSION`, we define the corresponding value of `MOD_VALUE`.

[C]

Listing 30: `defs.h`

```

1 #ifndef APP_VERSION
2 #define APP_VERSION      1
3 #endif
4
5 #if APP_VERSION == 1
6 #define MOD_VALUE        4
7 #endif
8
9 #if APP_VERSION == 2
10 #define MOD_VALUE       5
11 #endif

```

Listing 31: `main.c`

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "defs.h"
5
6 int func(int x)
7 {
8     return x % MOD_VALUE;
9 }
10
11 int main()
12 {
13     int a, b;
14
15     a = 10;
16     b = func(a);
17

```

(continues on next page)

(continued from previous page)

```
18     return 0;
19 }
```

If not defined outside, the code above will compile version #1 of the application. We can change this by specifying a value for APP_VERSION during the build (e.g. as a Makefile switch).

For the Ada version of this code, we can create two configuration packages for each version of the application. For example:

[Ada]

Listing 32: app_defs.ads

```
1 -- ./src/app_1/app_defs.ads
2
3 package App_Defs is
4
5     Mod_Value : constant Integer := 4;
6
7 end App_Defs;
```

Listing 33: func.ads

```
1 function Func (X : Integer) return Integer;
```

Listing 34: func.adb

```
1 with App_Defs;
2
3 function Func (X : Integer) return Integer is
4 begin
5     return X mod App_Defs.Mod_Value;
6 end Func;
```

Listing 35: main.adb

```
1 with Func;
2
3 procedure Main is
4     A, B : Integer;
5 begin
6     A := 10;
7     B := Func (A);
8 end Main;
```

Build output

```
main.adb:4:07: warning: variable "B" is assigned but never read [-gnatwm]
main.adb:7:04: warning: possibly useless assignment to "B", value might not be
 ↴ referenced [-gnatwm]
```

The code above shows the version #1 of the configuration package. The corresponding implementation for version #2 looks like this:

```
-- ./src/app_2/app_defs.ads
package App_Defs is
    Mod_Value : constant Integer := 5;
```

(continues on next page)

(continued from previous page)

```
end App_Defs;
```

Again, we just need to select the appropriate configuration package for each version of the build, which we can easily do when using **GPRbuild**.

68.3 Handling variability & reusability dynamically

68.3.1 Records with discriminants

In basic terms, records with discriminants are records that include "parameters" in their type definitions. This allows for adding more flexibility to the type definition. In the section about *pointers* (page 750), we've seen this example:

[Ada]

Listing 36: main.adb

```
1 procedure Main is
2   type Arr is array (Integer range <>) of Integer;
3
4   type S (Last : Positive) is record
5     A : Arr (0 .. Last);
6   end record;
7
8   V : S (9);
9 begin
10   null;
11 end Main;
```

Build output

```
main.adb:8:04: warning: variable "V" is never read and never assigned [-gnatwv]
```

Here, `Last` is the discriminant for type `S`. When declaring the variable `V` as `S (9)`, we specify the actual index of the last position of the array component `A` by setting the `Last` discriminant to 9.

We can create an equivalent implementation in C by declaring a **struct** with a pointer to an array:

[C]

Listing 37: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct {
5   int * a;
6   const int last;
7 } S;
8
9 S init_s (int last)
10 {
11   S v = { malloc (sizeof(int) * last + 1), last };
12   return v;
13 }
```

(continues on next page)

(continued from previous page)

```

14 int main(int argc, const char * argv[])
15 {
16     S v = init_s (9);
17
18     return 0;
19 }
```

Here, we need to explicitly allocate the a array of the S struct via a call to `malloc()`, which allocates memory space on the heap. In the Ada version, in contrast, the array (V.A) is allocated on the stack and we don't need to explicitly allocate it.

Note that the information that we provide as the discriminant to the record type (in the Ada code) is constant, so we cannot assign a value to it. For example, we cannot write:

[Ada]

```
V.Last := 10;      -- COMPILATION ERROR!
```

In the C version, we declare the `last` field constant to get the same behavior.

[C]

```
v.last = 10;      // COMPILATION ERROR!
```

Note that the information provided as discriminants is visible. In the example above, we could display `Last` by writing:

[Ada]

```
Put_Line ("Last : " & Integer'Image (V.Last));
```

Also note that, even if a type is private, we can still access the information of the discriminants if they are visible in the *public* part of the type declaration. Let's rewrite the example above:

[Ada]

Listing 38: array_definition.ads

```

1 package Array_Definition is
2     type Arr is array (Integer range <>) of Integer;
3
4     type S (Last : Integer) is private;
5
6     private
7         type S (Last : Integer) is record
8             A : Arr (0 .. Last);
9         end record;
10
11 end Array_Definition;
```

Listing 39: main.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Array_Definition; use Array_Definition;
3
4 procedure Main is
5     V : S (9);
6 begin
7     Put_Line ("Last : " & Integer'Image (V.Last));
8 end Main;
```

Build output

```
main.adb:5:04: warning: variable "V" is read but never assigned [-gnatwv]
```

Runtime output

```
Last : 9
```

Even though the S type is now private, we can still display Last because this discriminant is visible in the *non-private* part of package Array_Definition.

68.3.2 Variant records

In simple terms, a variant record is a record with discriminants that allows for changing its structure. Basically, it's a record containing a **case**. This is the general structure:

[Ada]

```
type Var_Rec (V : F) is record
    case V is
        when Opt_1 => F1 : Type_1;
        when Opt_2 => F2 : Type_2;
    end case;
end record;
```

Let's look at this example:

[Ada]

Listing 40: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      type Float_Int (Use_Float : Boolean) is record
6          case Use_Float is
7              when True  => F : Float;
8              when False => I : Integer;
9          end case;
10     end record;
11
12     procedure Display (V : Float_Int) is
13 begin
14     if V.Use_Float then
15         Put_Line ("Float value: " & Float'Image (V.F));
16     else
17         Put_Line ("Integer value: " & Integer'Image (V.I));
18     end if;
19 end Display;
20
21 F : constant Float_Int := (Use_Float => True,  F => 10.0);
22 I : constant Float_Int := (Use_Float => False, I => 9);
23
24 begin
25     Display (F);
26     Display (I);
27 end Main;
```

Runtime output

```
Float value: 1.00000E+01
Integer value: 9
```

Here, we declare F containing a floating-point value, and I containing an integer value. In the Display procedure, we present the correct information to the user according to the Use_Float discriminant of the Float_Int type.

We can implement this example in C by using unions:

[C]

Listing 41: main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct {
5     int use_float;
6     union {
7         float f;
8         int i;
9     };
10 } float_int;
11
12 float_int init_float (float f)
13 {
14     float_int v;
15
16     v.use_float = 1;
17     v.f          = f;
18     return v;
19 }
20
21 float_int init_int (int i)
22 {
23     float_int v;
24
25     v.use_float = 0;
26     v.i          = i;
27     return v;
28 }
29
30 void display (float_int v)
31 {
32     if (v.use_float) {
33         printf("Float value : %f\n", v.f);
34     }
35     else {
36         printf("Integer value : %d\n", v.i);
37     }
38 }
39
40 int main(int argc, const char * argv[])
41 {
42     float_int f = init_float (10.0);
43     float_int i = init_int (9);
44
45     display (f);
46     display (i);
47
48     return 0;
49 }
```

Runtime output

```
Float value : 10.000000
Integer value : 9
```

Similar to the Ada code, we declare `f` containing a floating-point value, and `i` containing an integer value. One difference is that we use the `init_float()` and `init_int()` functions to initialize the `float_int` struct. These functions initialize the correct field of the union and set the `use_float` field accordingly.

Variant records and unions

There is, however, a difference in accessibility between variant records in Ada and unions in C. In C, we're allowed to access any field of the union regardless of the initialization:

[C]

```
float_int v = init_float (10.0);
printf("Integer value : %d\n", v.i);
```

This feature is useful to create overlays. In this specific example, however, the information displayed to the user doesn't make sense, since the union was initialized with a floating-point value (`v.f`) and, by accessing the integer field (`v.i`), we're displaying it as if it was an integer value.

In Ada, accessing the wrong component would raise an exception at run-time ("discriminant check failed"), since the component is checked before being accessed:

[Ada]

```
V : constant Float_Int := (Use_Float => True, F => 10.0);
begin
  Put_Line ("Integer value: " & Integer'Image (V.I));
  -- ^ Constraint_Error is raised!
```

Using this method prevents wrong information being used in other parts of the program.

To get the same behavior in Ada as we do in C, we need to explicitly use the `Unchecked_Union` aspect in the type declaration. This is the modified example:

[Ada]

Listing 42: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5    type Float_Int_Union (Use_Float : Boolean) is record
6      case Use_Float is
7        when True  => F : Float;
8        when False => I : Integer;
9      end case;
10     end record
11     with Unchecked_Union;
12
13   V : constant Float_Int_Union := (Use_Float => True, F => 10.0);
14
15 begin
16   Put_Line ("Integer value: " & Integer'Image (V.I));
17 end Main;
```

Runtime output

```
Integer value: 1092616192
```

Now, we can display the integer component (V.I) even though we initialized the floating-point component (V.F). As expected, the information displayed by the test application in this case doesn't make sense.

Note that, when using the `Unchecked_Union` aspect in the declaration of a variant record, the reference discriminant is not available anymore, since it isn't stored as part of the record. Therefore, we cannot access the `Use_Float` discriminant as in the following code:

[Ada]

```
V : constant Float_Int_Union := (Use_Float => True, F => 10.0);
begin
  if V.Use_Float then      -- COMPILATION ERROR!
    -- Do something...
  end if;
```

Unchecked unions are particularly useful in Ada when creating bindings for C code.

Optional components

We can also use variant records to specify optional components of a record. For example:

[Ada]

Listing 43: main.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Main is
4    type Arr is array (Integer range <>) of Integer;
5
6    type Extra_Info is (No, Yes);
7
8    type S_Var (Last : Integer; Has_Extra_Info : Extra_Info) is record
9      A : Arr (0 .. Last);
10
11      case Has_Extra_Info is
12        when No  => null;
13        when Yes => B : Arr (0 .. Last);
14      end case;
15    end record;
16
17    V1 : S_Var (Last => 9, Has_Extra_Info => Yes);
18    V2 : S_Var (Last => 9, Has_Extra_Info => No);
19 begin
20   Put_Line ("Size of V1 is: " & Integer'Image (V1'Size));
21   Put_Line ("Size of V2 is: " & Integer'Image (V2'Size));
22 end Main;
```

Build output

```
main.adb:17:04: warning: variable "V1" is read but never assigned [-gnatwv]
main.adb:18:04: warning: variable "V2" is read but never assigned [-gnatwv]
```

Runtime output

```
Size of V1 is: 704
Size of V2 is: 384
```

Here, in the declaration of `S_Var`, we don't have any component in case `Has_Extra_Info` is false. The component is simply set to `null` in this case.

When running the example above, we see that the size of `V1` is greater than the size of `V2` due to the extra `B` component — which is only included when `Has_Extra_Info` is true.

Optional output information

We can use optional components to prevent subprograms from generating invalid information that could be misused by the caller. Consider the following example:

[C]

Listing 44: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float calculate (float f1,
5                 float f2,
6                 int *success)
7 {
8     if (f1 < f2) {
9         *success = 1;
10        return f2 - f1;
11    }
12    else {
13        *success = 0;
14        return 0.0;
15    }
16 }
17
18 void display (float v,
19                int success)
20 {
21     if (success) {
22         printf("Value = %f\n", v);
23     }
24     else {
25         printf("Calculation error!\n");
26     }
27 }
28
29 int main(int argc, const char * argv[])
30 {
31     float f;
32     int success;
33
34     f = calculate (1.0, 0.5, &success);
35     display (f, success);
36
37     f = calculate (0.5, 1.0, &success);
38     display (f, success);
39
40     return 0;
41 }
```

Runtime output

```
Calculation error!
Value = 0.500000
```

In this code, we're using the output parameter `success` of the `calculate()` function to indicate whether the calculation was successful or not. This approach has a major problem: there's no way to prevent that the invalid value returned by `calculate()` in case of an error is misused in another computation. For example:

[C]

```
int main(int argc, const char * argv[])
{
    float f;
    int success;

    f = calculate (1.0, 0.5, &success);

    f = f * 0.25; // Using f in another computation even though
                   // calculate() returned a dummy value due to error!
                   // We should have evaluated "success", but we didn't.

    return 0;
}
```

We cannot prevent access to the returned value or, at least, force the caller to evaluate `success` before using the returned value.

This is the corresponding code in Ada:

[Ada]

Listing 45: main.adb

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      function Calculate (F1, F2 : Float;
6                          Success : out Boolean) return Float is
7  begin
8      if F1 < F2 then
9          Success := True;
10         return F2 - F1;
11     else
12         Success := False;
13         return 0.0;
14     end if;
15  end Calculate;
16
17  procedure Display (V : Float; Success : Boolean) is
18  begin
19      if Success then
20          Put_Line ("Value = " & Float'Image (V));
21      else
22          Put_Line ("Calculation error!");
23      end if;
24  end Display;
25
26  F      : Float;
27  Success : Boolean;
28  begin
29      F := Calculate (1.0, 0.5, Success);
30      Display (F, Success);
31
32      F := Calculate (0.5, 1.0, Success);
33      Display (F, Success);
34  end Main;
```

Runtime output

```
Calculation error!
Value = 5.00000E-01
```

The Ada code above suffers from the same drawbacks as the C code. Again, there's no way to prevent misuse of the invalid value returned by Calculate in case of errors.

However, in Ada, we can use variant records to make the component unavailable and therefore prevent misuse of this information. Let's rewrite the original example and wrap the returned value in a variant record:

[Ada]

Listing 46: main.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Main is
4
5    type Opt_Float (Success : Boolean) is record
6      case Success is
7        when False => null;
8        when True   => F : Float;
9      end case;
10     end record;
11
12    function Calculate (F1, F2 : Float) return Opt_Float is
13    begin
14      if F1 < F2 then
15        return (Success => True, F => F2 - F1);
16      else
17        return (Success => False);
18      end if;
19    end Calculate;
20
21    procedure Display (V : Opt_Float) is
22    begin
23      if V.Success then
24        Put_Line ("Value = " & Float'Image (V.F));
25      else
26        Put_Line ("Calculation error!");
27      end if;
28    end Display;
29
30  begin
31    Display (Calculate (1.0, 0.5));
32    Display (Calculate (0.5, 1.0));
33  end Main;
```

Runtime output

```
Calculation error!
Value = 5.00000E-01
```

In this example, we can determine whether the calculation was successful or not by evaluating the Success component of the Opt_Float. If the calculation wasn't successful, we won't be able to access the F component of the Opt_Float. As mentioned before, trying to access the component in this case would raise an exception. Therefore, in case of errors, we can ensure that no information is misused after the call to Calculate.

68.3.3 Object orientation

In the [previous section](#) (page 785), we've seen that we can add variability to records by using discriminants. Another approach is to use *tagged* records, which are the base for object-oriented programming in Ada.

Type extension

A tagged record type is declared by adding the **tagged** keyword. For example:

[Ada]

Listing 47: main.adb

```

1  procedure Main is
2
3      type Rec is record
4          V : Integer;
5      end record;
6
7      type Tagged_Rec is tagged record
8          V : Integer;
9      end record;
10
11     R1 : Rec;
12     R2 : Tagged_Rec;
13
14     pragma Unreferenced (R1, R2);
15 begin
16     R1 := (V => 0);
17     R2 := (V => 0);
18 end Main;
```

In this simple example, there isn't much difference between the `Rec` and `Tagged_Rec` type. However, tagged types can be derived and extended. For example:

[Ada]

Listing 48: main.adb

```

1  procedure Main is
2
3      type Rec is record
4          V : Integer;
5      end record;
6
7      -- We cannot declare this:
8
9      -- type Ext_Rec is new Rec with record
10     --   V : Integer;
11     -- end record;
12
13      type Tagged_Rec is tagged record
14          V : Integer;
15      end record;
16
17      -- But we can declare this:
18
19      type Ext_Tagged_Rec is new Tagged_Rec with record
20          V2 : Integer;
21      end record;
```

(continues on next page)

(continued from previous page)

```

22      R1 : Rec;
23      R2 : Tagged_Rec;
24      R3 : Ext_Tagged_Rec;
25
26      pragma Unreferenced (R1, R2, R3);
27  begin
28      R1 := (V => 0);
29      R2 := (V => 0);
30      R3 := (V => 0, V2 => 0);
31  end Main;

```

As indicated in the example, a type derived from an untagged type cannot have an extension. The compiler indicates this error if you uncomment the declaration of the Ext_Rec type above. In contrast, we can extend a tagged type, as we did in the declaration of Ext_Tagged_Rec. In this case, Ext_Tagged_Rec has all the components of the Tagged_Rec type (V, in this case) plus the additional components from its own type declaration (V2, in this case).

Overriding subprograms

Previously, we've seen that subprograms can be overridden. For example, if we had implemented a Reset and a Display procedure for the Rec type that we declared above, these procedures would be available for an Ext_Rec type derived from Rec. Also, we could override these procedures for the Ext_Rec type. In Ada, we don't need object-oriented programming features to do that: simple (untagged) records can be used to derive types, inherit operations and override them. However, in applications where the actual subprogram to be called is determined dynamically at run-time, we need dispatching calls. In this case, we must use tagged types to implement this.

Comparing untagged and tagged types

Let's discuss the similarities and differences between untagged and tagged types based on this example:

[Ada]

Listing 49: p.ads

```

1  package P is
2
3      type Rec is record
4          V : Integer;
5      end record;
6
7      procedure Display (R : Rec);
8      procedure Reset (R : out Rec);
9
10     type New_Rec is new Rec;
11
12     overriding procedure Display (R : New_Rec);
13     not overriding procedure New_Op (R : in out New_Rec);
14
15     type Tagged_Rec is tagged record
16         V : Integer;
17     end record;
18

```

(continues on next page)

(continued from previous page)

```

19  procedure Display (R : Tagged_Rec);
20  procedure Reset (R : out Tagged_Rec);
21
22  type Ext_Tagged_Rec is new Tagged_Rec with record
23      V2 : Integer;
24  end record;
25
26  overriding procedure Display (R : Ext_Tagged_Rec);
27  overriding procedure Reset (R : out Ext_Tagged_Rec);
28  not overriding procedure New_Op (R : in out Ext_Tagged_Rec);
29
30 end P;

```

Listing 50: p.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body P is
4
5      procedure Display (R : Rec) is
6      begin
7          Put_Line ("TYPE: REC");
8          Put_Line ("Rec.V = " & Integer'Image (R.V));
9          New_Line;
10     end Display;
11
12     procedure Reset (R : out Rec) is
13     begin
14         R.V := 0;
15     end Reset;
16
17     procedure Display (R : New_Rec) is
18     begin
19         Put_Line ("TYPE: NEW_REC");
20         Put_Line ("New_Rec.V = " & Integer'Image (R.V));
21         New_Line;
22     end Display;
23
24     procedure New_Op (R : in out New_Rec) is
25     begin
26         R.V := R.V + 1;
27     end New_Op;
28
29     procedure Display (R : Tagged_Rec) is
30     begin
31         -- Using External_Tag attribute to retrieve the tag as a string
32         Put_Line ("TYPE: " & Tagged_Rec'External_Tag);
33         Put_Line ("Tagged_Rec.V = " & Integer'Image (R.V));
34         New_Line;
35     end Display;
36
37     procedure Reset (R : out Tagged_Rec) is
38     begin
39         R.V := 0;
40     end Reset;
41
42     procedure Display (R : Ext_Tagged_Rec) is
43     begin
44         -- Using External_Tag attribute to retrieve the tag as a string
45         Put_Line ("TYPE: " & Ext_Tagged_Rec'External_Tag);
46         Put_Line ("Ext_Tagged_Rec.V = " & Integer'Image (R.V));

```

(continues on next page)

(continued from previous page)

```

47 Put_Line ("Ext_Tagged_Rec.V2 = " & Integer'Image (R.V2));
48 New_Line;
49 end Display;

50
51 procedure Reset (R : out Ext_Tagged_Rec) is
52 begin
53     Tagged_Rec (R).Reset;
54     R.V2 := 0;
55 end Reset;

56
57 procedure New_Op (R : in out Ext_Tagged_Rec) is
58 begin
59     R.V := R.V + 1;
60 end New_Op;

61
62 end P;

```

Listing 51: main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with P;           use P;

3
4 procedure Main is
5     X_Rec          : Rec;
6     X_New_Rec      : New_Rec;
7
8     X_Tagged_Rec   : aliased Tagged_Rec;
9     X_Ext_Tagged_Rec : aliased Ext_Tagged_Rec;
10
11    X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class
12        := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
13 begin
14
15     -- Reset all objects
16
17     Reset (X_Rec);
18     Reset (X_New_Rec);
19     X_Tagged_Rec.Reset;    -- we could write "Reset (X_Tagged_Rec)" as well
20     X_Ext_Tagged_Rec.Reset;
21
22
23     -- Use new operations when available
24
25     New_Op (X_New_Rec);
26     X_Ext_Tagged_Rec.New_Op;
27
28
29     -- Display all objects
30
31     Display (X_Rec);
32     Display (X_New_Rec);
33     X_Tagged_Rec.Display; -- we could write "Display (X_Tagged_Rec)" as well
34     X_Ext_Tagged_Rec.Display;
35
36
37     -- Resetting and display objects of Tagged_Rec'Class
38
39     Put_Line ("Operations on Tagged_Rec'Class");
40     Put_Line ("-----");
41     for E of X_Tagged_Rec_Array loop
42         E.Reset;

```

(continues on next page)

(continued from previous page)

```
43      E.Display;
44  end loop;
45 end Main;
```

Runtime output

```
TYPE: REC
Rec.V = 0

TYPE: NEW_REC
New_Rec.V = 1

TYPE: P.TAGGED_REC
Tagged_Rec.V = 0

TYPE: P.EXT_TAGGED_REC
Ext_Tagged_Rec.V = 1
Ext_Tagged_Rec.V2 = 0

Operations on Tagged_Rec'Class
-----
TYPE: P.TAGGED_REC
Tagged_Rec.V = 0

TYPE: P.EXT_TAGGED_REC
Ext_Tagged_Rec.V = 0
Ext_Tagged_Rec.V2 = 0
```

These are the similarities between untagged and tagged types:

- We can derive types and inherit operations in both cases.
 - Both X_New_Rec and X_Ext_Tagged_Rec inherit the Display and Reset procedures from their respective ancestors.
- We can override operations in both cases.
- We can implement new operations in both cases.
 - Both X_New_Rec and X_Ext_Tagged_Rec implement a procedure called New_Op, which is not available for their respective ancestors.

Now, let's look at the differences between untagged and tagged types:

- We can dispatch calls for a given type class.
 - This is what we do when we iterate over objects of the Tagged_Rec class — in the loop over X_Tagged_Rec_Array at the last part of the Main procedure.
- We can use the dot notation.
 - We can write both E.Reset or Reset (E) forms: they're equivalent.

Dispatching calls

Let's look more closely at the dispatching calls implemented above. First, we declare the X_Tagged_Rec_Array array and initialize it with the access to objects of both parent and derived tagged types:

[Ada]

```
X_Tagged_Rec      : aliased Tagged_Rec;
X_Ext_Tagged_Rec : aliased Ext_Tagged_Rec;

X_Tagged_Rec_Array : constant array (1 .. 2) of access Tagged_Rec'Class
                           := (X_Tagged_Rec'Access, X_Ext_Tagged_Rec'Access);
```

Here, we use the **aliased** keyword to be able to get access to the objects (via the '**Access** attribute).

Then, we loop over this array and call the Reset and Display procedures:

[Ada]

```
for E of X_Tagged_Rec_Array loop
  E.Reset;
  E.Display;
end loop;
```

Since we're using dispatching calls, the actual procedure that is selected depends on the type of the object. For the first element (`X_Tagged_Rec_Array (1)`), this is `Tagged_Rec`, while for the second element (`X_Tagged_Rec_Array (2)`), this is `Ext_Tagged_Rec`.

Dispatching calls are only possible for a type class — for example, the `Tagged_Rec'Class`. When the type of an object is known at compile time, the calls won't dispatch at runtime. For example, the call to the `Reset` procedure of the `X_Ext_Tagged_Rec` object (`X_Ext_Tagged_Rec.Reset`) will always take the overridden `Reset` procedure of the `Ext_Tagged_Rec` type. Similarly, if we perform a view conversion by writing `Tagged_Rec (A_Ext_Tagged_Rec).Display`, we're instructing the compiler to interpret `A_Ext_Tagged_Rec` as an object of type `Tagged_Rec`, so that the compiler selects the `Display` procedure of the `Tagged_Rec` type.

Interfaces

Another useful feature of object-oriented programming is the use of interfaces. In this case, we can define abstract operations, and implement them in the derived tagged types. We declare an interface by simply writing **type T is interface**. For example:

[Ada]

```
type My_Interface is interface;

procedure Op (Obj : My_Interface) is abstract;

-- We cannot declare actual objects of an interface:
-- 
-- Obj : My_Interface; -- ERROR!
```

All operations on an interface type are abstract, so we need to write **is abstract** in the signature — as we did in the declaration of `Op` above. Also, since interfaces are abstract types and don't have an actual implementation, we cannot declare objects for it.

We can derive tagged types from an interface and implement the actual operations of that interface:

[Ada]

```
type My_Derived is new My_Interface with null record;
procedure Op (Obj : My_Derived);
```

Note that we're not using the **tagged** keyword in the declaration because any type derived from an interface is automatically tagged.

Let's look at an example with an interface and two derived tagged types:

[Ada]

Listing 52: p.ads

```
1 package P is
2
3     type Display_Interface is interface;
4     procedure Display (D : Display_Interface) is abstract;
5
6     type Small_Display_Type is new Display_Interface with null record;
7     procedure Display (D : Small_Display_Type);
8
9     type Big_Display_Type is new Display_Interface with null record;
10    procedure Display (D : Big_Display_Type);
11
12 end P;
```

Listing 53: p.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body P is
4
5     procedure Display (D : Small_Display_Type) is
6         pragma Unreferenced (D);
7     begin
8         Put_Line ("Using Small_Display_Type");
9     end Display;
10
11    procedure Display (D : Big_Display_Type) is
12        pragma Unreferenced (D);
13    begin
14        Put_Line ("Using Big_Display_Type");
15    end Display;
16
17 end P;
```

Listing 54: main.adb

```
1 with P; use P;
2
3 procedure Main is
4     D_Small : Small_Display_Type;
5     D_Big   : Big_Display_Type;
6
7     procedure Dispatching_Display (D : Display_Interface'Class) is
8     begin
9         D.Display;
10    end Dispatching_Display;
11
12 begin
```

(continues on next page)

(continued from previous page)

```

13     Dispatching_Display (D_Small);
14     Dispatching_Display (D_Big);
15 end Main;

```

Runtime output

```

Using Small_Display_Type
Using Big_Display_Type

```

In this example, we have an interface type `Display_Interface` and two tagged types that are derived from `Display_Interface`: `Small_Display_Type` and `Big_Display_Type`.

Both types (`Small_Display_Type` and `Big_Display_Type`) implement the interface by overriding the `Display` procedure. Then, in the inner procedure `Dispatching_Display` of the `Main` procedure, we perform a dispatching call depending on the actual type of `D`.

Deriving from multiple interfaces

We may derive a type from multiple interfaces by simply writing `type Derived_T is new T1 and T2 with null record`. For example:

[Ada]

Listing 55: transceivers.ads

```

1 package Transceivers is
2
3     type Send_Interface is interface;
4
5     procedure Send (Obj : in out Send_Interface) is abstract;
6
7     type Receive_Interface is interface;
8
9     procedure Receive (Obj : in out Receive_Interface) is abstract;
10
11    type Transceiver is new Send_Interface and Receive_Interface
12        with null record;
13
14    procedure Send (D : in out Transceiver);
15    procedure Receive (D : in out Transceiver);
16
17 end Transceivers;

```

Listing 56: transceivers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Transceivers is
4
5     procedure Send (D : in out Transceiver) is
6         pragma Unreferenced (D);
7     begin
8         Put_Line ("Sending data...");
9     end Send;
10
11    procedure Receive (D : in out Transceiver) is
12        pragma Unreferenced (D);
13    begin
14        Put_Line ("Receiving data...");

```

(continues on next page)

(continued from previous page)

```

15    end Receive;
16
17 end Transceivers;
```

Listing 57: main.adb

```

1 with Transceivers; use Transceivers;
2
3 procedure Main is
4     D : Transceiver;
5 begin
6     D.Send;
7     D.Receive;
8 end Main;
```

Runtime output

```

Sending data...
Receiving data...
```

In this example, we're declaring two interfaces (Send_Interface and Receive_Interface) and the tagged type Transceiver that derives from both interfaces. Since we need to implement the interfaces, we implement both Send and Receive for Transceiver.

Abstract tagged types

We may also declare abstract tagged types. Note that, because the type is abstract, we cannot use it to declare objects for it — this is the same as for interfaces. We can only use it to derive other types. Let's look at the abstract tagged type declared in the Abstract_Transceivers package:

[Ada]

Listing 58: abstract_transceivers.ads

```

1 with Transceivers; use Transceivers;
2
3 package Abstract_Transceivers is
4
5     type Abstract_Transceiver is abstract new Send_Interface and
6         Receive_Interface with null record;
7
8     procedure Send (D : in out Abstract_Transceiver);
9     -- We don't implement Receive for Abstract_Transceiver!
10
11 end Abstract_Transceivers;
```

Listing 59: abstract_transceivers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Abstract_Transceivers is
4
5     procedure Send (D : in out Abstract_Transceiver) is
6         pragma Unreferenced (D);
7     begin
8         Put_Line ("Sending data...");
9     end Send;
```

(continues on next page)

(continued from previous page)

```

10
11 end Abstract_Transceivers;
```

Listing 60: main.adb

```

1 with Abstract_Transceivers; use Abstract_Transceivers;
2
3 procedure Main is
4   D : Abstract_Transceiver;
5 begin
6   D.Send;
7   D.Receive;
8 end Main;
```

Build output

```

main.adb:4:09: error: type of object cannot be abstract
main.adb:7:06: error: call to abstract procedure must be dispatching
gprbuild: *** compilation phase failed
```

In this example, we declare the abstract tagged type `Abstract_Transceiver`. Here, we're only partially implementing the interfaces from which this type is derived: we're implementing `Send`, but we're skipping the implementation of `Receive`. Therefore, `Receive` is an abstract operation of `Abstract_Transceiver`. Since any tagged type that has abstract operations is abstract, we must indicate this by adding the `abstract` keyword in type declaration.

Also, when compiling this example, we get an error because we're trying to declare an object of `Abstract_Transceiver` (in the `Main` procedure), which is not possible. Naturally, if we derive another type from `Abstract_Transceiver` and implement `Receive` as well, then we can declare objects of this derived type. This is what we do in the `Full_Transceivers` below:

[Ada]

Listing 61: full_transceivers.ads

```

1 with Abstract_Transceivers; use Abstract_Transceivers;
2
3 package Full_Transceivers is
4
5   type Full_Transceiver is new Abstract_Transceiver with null record;
6   procedure Receive (D : in out Full_Transceiver);
7
8 end Full_Transceivers;
```

Listing 62: full_transceivers.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Full_Transceivers is
4
5   procedure Receive (D : in out Full_Transceiver) is
6     pragma Unreferenced (D);
7   begin
8     Put_Line ("Receiving data...");
9   end Receive;
10
11 end Full_Transceivers;
```

Listing 63: main.adb

```
1 with Full_Transceivers; use Full_Transceivers;
2
3 procedure Main is
4   D : Full_Transceiver;
5 begin
6   D.Send;
7   D.Receive;
8 end Main;
```

Runtime output

```
Sending data...
Receiving data...
```

Here, we implement the Receive procedure for the Full_Transceiver. Therefore, the type doesn't have any abstract operation, so we can use it to declare objects.

From simple derivation to OOP

In the [section about simple derivation](#) (page 774), we've seen an example where the actual selection was done at *implementation* time by renaming one of the packages:

[Ada]

```
with Drivers_1;

package Drivers renames Drivers_1;
```

Although this approach is useful in many cases, there might be situations where we need to select the actual driver dynamically at runtime. Let's look at how we could rewrite that example using interfaces, tagged types and dispatching calls:

[Ada]

Listing 64: drivers_base.ads

```
1 package Drivers_Base is
2
3   type Transceiver is interface;
4
5   procedure Send (Device : Transceiver; Data : Integer) is abstract;
6   procedure Receive (Device : Transceiver; Data : out Integer) is abstract;
7   procedure Display (Device : Transceiver) is abstract;
8
9 end Drivers_Base;
```

Listing 65: drivers_1.ads

```
1 with Drivers_Base;
2
3 package Drivers_1 is
4
5   type Transceiver is new Drivers_Base.Transceiver with null record;
6
7   procedure Send (Device : Transceiver; Data : Integer);
8   procedure Receive (Device : Transceiver; Data : out Integer);
9   procedure Display (Device : Transceiver);
```

(continues on next page)

(continued from previous page)

```

10
11 end Drivers_1;
```

Listing 66: drivers_1.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Drivers_1 is
4
5   procedure Send (Device : Transceiver; Data : Integer) is null;
6
7   procedure Receive (Device : Transceiver; Data : out Integer) is
8     pragma Unreferenced (Device);
9   begin
10    Data := 42;
11  end Receive;
12
13  procedure Display (Device : Transceiver) is
14    pragma Unreferenced (Device);
15  begin
16    Put_Line ("Using Drivers_1");
17  end Display;
18
19 end Drivers_1;
```

Listing 67: drivers_2.ads

```

1 with Drivers_Base;
2
3 package Drivers_2 is
4
5   type Transceiver is new Drivers_Base.Transceiver with null record;
6
7   procedure Send (Device : Transceiver; Data : Integer);
8   procedure Receive (Device : Transceiver; Data : out Integer);
9   procedure Display (Device : Transceiver);
10
11 end Drivers_2;
```

Listing 68: drivers_2.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Drivers_2 is
4
5   procedure Send (Device : Transceiver; Data : Integer) is null;
6
7   procedure Receive (Device : Transceiver; Data : out Integer) is
8     pragma Unreferenced (Device);
9   begin
10    Data := 7;
11  end Receive;
12
13  procedure Display (Device : Transceiver) is
14    pragma Unreferenced (Device);
15  begin
16    Put_Line ("Using Drivers_2");
17  end Display;
18
19 end Drivers_2;
```

Listing 69: main.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  with Drivers_Base;
4  with Drivers_1;
5  with Drivers_2;
6
7  procedure Main is
8      D1 : aliased Drivers_1.Transceiver;
9      D2 : aliased Drivers_2.Transceiver;
10     D : access Drivers_Base.Transceiver'Class;
11
12    I : Integer;
13
14    type Driver_Number is range 1 .. 2;
15
16    procedure Select_Driver (N : Driver_Number) is
17    begin
18        if N = 1 then
19            D := D1'Access;
20        else
21            D := D2'Access;
22        end if;
23        D.Display;
24    end Select_Driver;
25
26 begin
27    Select_Driver (1);
28    D.Send (999);
29    D.Receive (I);
30    Put_Line (Integer'Image (I));
31
32    Select_Driver (2);
33    D.Send (999);
34    D.Receive (I);
35    Put_Line (Integer'Image (I));
36 end Main;
```

Runtime output

```
Using Drivers_1
42
Using Drivers_2
7
```

In this example, we declare the `Transceiver` interface in the `Drivers_Base` package. This interface is then used to derive the tagged types `Transceiver` from both `Drivers_1` and `Drivers_2` packages.

In the `Main` procedure, we use the access to `Transceiver'Class` — from the interface declared in the `Drivers_Base` package — to declare `D`. This object `D` contains the access to the actual driver loaded at any specific time. We select the driver at runtime in the inner `Select_Driver` procedure, which initializes `D` (with the access to the selected driver). Then, any operation on `D` triggers a dispatching call to the selected driver.

Further resources

In the appendices, we have a step-by-step *hands-on overview of object-oriented programming* (page 837) that discusses how to translate a simple system written in C to an equivalent system in Ada using object-oriented programming.

68.3.4 Pointer to subprograms

Pointers to subprograms allow us to dynamically select an appropriate subprogram at run-time. This selection might be triggered by an external event, or simply by the user. This can be useful when multiple versions of a routine exist, and the decision about which one to use cannot be made at compilation time.

This is an example on how to declare and use pointers to functions in C:

[C]

Listing 70: main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void show_msg_v1 (char *msg)
5 {
6     printf("Using version #1: %s\n", msg);
7 }
8
9 void show_msg_v2 (char *msg)
10 {
11     printf("Using version #2:\n %s\n", msg);
12 }
13
14 int main()
15 {
16     int selection = 1;
17     void (*current_show_msg) (char *);
18
19     switch (selection)
20     {
21         case 1: current_show_msg = &show_msg_v1; break;
22         case 2: current_show_msg = &show_msg_v2; break;
23         default: current_show_msg = NULL; break;
24     }
25
26     if (current_show_msg != NULL)
27     {
28         current_show_msg ("Hello there!");
29     }
30     else
31     {
32         printf("ERROR: no version of show_msg() selected!\n");
33     }
34
35     return 0;
36 }
```

Runtime output

Using version #1: Hello there!

The example above contains two versions of the `show_msg()` function: `show_msg_v1()`

and `show_msg_v2()`. The function is selected depending on the value of `selection`, which initializes the function pointer `current_show_msg`. If there's no corresponding value, `current_show_msg` is set to `null` — alternatively, we could have selected a default version of `show_msg()` function. By calling `current_show_msg ("Hello there!")`, we're calling the function that `current_show_msg` is pointing to.

This is the corresponding implementation in Ada:

[Ada]

Listing 71: `show_subprogram_selection.adb`

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Show_Subprogram_Selection is
4
5      procedure Show_Msg_V1 (Msg : String) is
6      begin
7          Put_Line ("Using version #1: " & Msg);
8      end Show_Msg_V1;
9
10     procedure Show_Msg_V2 (Msg : String) is
11     begin
12         Put_Line ("Using version #2: ");
13         Put_Line (Msg);
14     end Show_Msg_V2;
15
16     type Show_Msg_Proc is access procedure (Msg : String);
17
18     Current_Show_Msg : Show_Msg_Proc;
19     Selection        : Natural;
20
21 begin
22     Selection := 1;
23
24     case Selection is
25         when 1      => Current_Show_Msg := Show_Msg_V1'Access;
26         when 2      => Current_Show_Msg := Show_Msg_V2'Access;
27         when others => Current_Show_Msg := null;
28     end case;
29
30     if Current_Show_Msg /= null then
31         Current_Show_Msg ("Hello there!");
32     else
33         Put_Line ("ERROR: no version of Show_Msg selected!");
34     end if;
35
36 end Show_Subprogram_Selection;
```

Runtime output

```
Using version #1: Hello there!
```

The structure of the code above is very similar to the one used in the C code. Again, we have two versions of `Show_Msg`: `Show_Msg_V1` and `Show_Msg_V2`. We set `Current_Show_Msg` according to the value of `Selection`. Here, we use `'Access` to get access to the corresponding procedure. If no version of `Show_Msg` is available, we set `Current_Show_Msg` to `null`.

Pointers to subprograms are also typically used as callback functions. This approach is extensively used in systems that process events, for example. Here, we could have a two-layered system:

- A layer of the system (an event manager) triggers events depending on information

from sensors.

- For each event, callback functions can be registered.
- The event manager calls registered callback functions when an event is triggered.
- Another layer of the system registers callback functions for specific events and decides what to do when those events are triggered.

This approach promotes information hiding and component decoupling because:

- the layer of the system responsible for managing events doesn't need to know what the callback function actually does, while
- the layer of the system that implements callback functions remains agnostic to implementation details of the event manager — for example, how events are implemented in the event manager.

Let's see an example in C where we have a `process_values()` function that calls a callback function (`process_one`) to process a list of values:

[C]

Listing 72: `process_values.h`

```

1 typedef int (*process_one_callback) (int);
2
3 void process_values (int *values,
4 int len,
5 process_one_callback process_one);
```

Listing 73: `process_values.c`

```

1 #include "process_values.h"
2
3 #include <assert.h>
4 #include <stdio.h>
5
6 void process_values (int *values,
7 int len,
8 process_one_callback process_one)
9 {
10     int i;
11
12     assert (process_one != NULL);
13
14     for (i = 0; i < len; i++)
15     {
16         values[i] = process_one (values[i]);
17     }
18 }
```

Listing 74: `main.c`

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "process_values.h"
5
6 int proc_10 (int val)
7 {
8     return val + 10;
9 }
```

(continues on next page)

(continued from previous page)

```

11 # define LEN_VALUES      5
12
13 int main()
14 {
15
16     int values[LEN_VALUES] = { 1, 2, 3, 4, 5 };
17     int i;
18
19     process_values (values, LEN_VALUES, &proc_10);
20
21     for (i = 0; i < LEN_VALUES; i++)
22     {
23         printf("Value [ %d ] = %d\n", i, values[i]);
24     }
25
26     return 0;
27 }
```

Runtime output

```

Value [0] = 11
Value [1] = 12
Value [2] = 13
Value [3] = 14
Value [4] = 15
```

As mentioned previously, `process_values()` doesn't have any knowledge about what `process_one()` does with the integer value it receives as a parameter. Also, we could replace `proc_10()` by another function without having to change the implementation of `process_values()`.

Note that `process_values()` calls an `assert()` for the function pointer to compare it against `null`. Here, instead of checking the validity of the function pointer, we're expecting the caller of `process_values()` to provide a valid pointer.

This is the corresponding implementation in Ada:

[Ada]

Listing 75: `values_processing.ads`

```

1 package Values_Processing is
2
3     type Integer_Array is array (Positive range <>) of Integer;
4
5     type Process_One_Callback is not null access
6         function (Value : Integer) return Integer;
7
8     procedure Process_Values (Values      : in out Integer_Array;
9                             Process_One :          Process_One_Callback);
10
11 end Values_Processing;
```

Listing 76: `values_processing.adb`

```

1 package body Values_Processing is
2
3     procedure Process_Values (Values      : in out Integer_Array;
4                               Process_One :          Process_One_Callback) is
5     begin
6         for I in Values'Range loop
```

(continues on next page)

(continued from previous page)

```

7      Values (I) := Process_One (Values (I));
8  end loop;
9 end Process_Values;
10
11 end Values_Processing;
```

Listing 77: proc_10.ads

```
1 function Proc_10 (Value : Integer) return Integer;
```

Listing 78: proc_10.adb

```

1 function Proc_10 (Value : Integer) return Integer is
2 begin
3     return Value + 10;
4 end Proc_10;
```

Listing 79: show_callback.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Values_Processing; use Values_Processing;
4 with Proc_10;
5
6 procedure Show_Callback is
7     Values : Integer_Array := (1, 2, 3, 4, 5);
8 begin
9     Process_Values (Values, Proc_10'Access);
10
11    for I in Values'Range loop
12        Put_Line ("Value [" & Positive'Image (I)
13                                & "] = "
14                                & Integer'Image (Values (I)));
15    end loop;
16 end Show_Callback;
```

Runtime output

```

Value [ 1] = 11
Value [ 2] = 12
Value [ 3] = 13
Value [ 4] = 14
Value [ 5] = 15
```

Similar to the implementation in C, the Process_Values procedure receives the access to a callback routine, which is then called for each value of the Values array.

Note that the declaration of Process_One_Callback makes use of the **not null access** declaration. By using this approach, we ensure that any parameter of this type has a valid value, so we can always call the callback routine.

68.4 Design by components using dynamic libraries

In the previous sections, we have shown how to use packages to create separate components of a system. As we know, when designing a complex system, it is advisable to separate concerns into distinct units, so we can use Ada packages to represent each unit of a system. In this section, we go one step further and create separate dynamic libraries for each component, which we'll then link to the main application.

Let's suppose we have a main system (Main_System) and a component A (Component_A) that we want to use in the main system. For example:

[Ada]

Listing 80: component_a.ads

```

1  --
2  -- File: component_a.ads
3  --
4  package Component_A is
5
6    type Float_Array is array (Positive range <>) of Float;
7
8    function Average (Data : Float_Array) return Float;
9
10 end Component_A;
```

Listing 81: component_a.adb

```

1  --
2  -- File: component_a.adb
3  --
4  package body Component_A is
5
6    function Average (Data : Float_Array) return Float is
7      Total : Float := 0.0;
8    begin
9      for Value of Data loop
10        Total := Total + Value;
11      end loop;
12      return Total / Float (Data'Length);
13    end Average;
14
15 end Component_A;
```

Listing 82: main_system.adb

```

1  --
2  -- File: main_system.adb
3  --
4  with Ada.Text_IO; use Ada.Text_IO;
5
6  with Component_A; use Component_A;
7
8  procedure Main_System is
9    Values      : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
10   Average_Value : Float;
11
12 begin
13   Average_Value := Average (Values);
14   Put_Line ("Average = " & Float'Image (Average_Value));
15 end Main_System;
```

Runtime output

```
Average = 1.15000E+01
```

Note that, in the source-code example above, we're indicating the name of each file. We'll now see how to organize those files in a structure that is suitable for the GNAT build system (**GPRbuild**).

In order to discuss how to create dynamic libraries, we need to dig into some details about the build system. With GNAT, we can use project files for **GPRbuild** to easily design dynamic libraries. Let's say we use the following directory structure for the code above:

```

|- component_a
  |   component_a.gpr
  |   |- src
  |     |   component_a.adb
  |     |   component_a.ads
|- main_system
  |   main_system.gpr
  |   |- src
  |     |   main_system.adb

```

Here, we have two directories: *component_a* and *main_system*. Each directory contains a project file (with the *.gpr* file extension) and a source-code directory (*src*).

In the source-code example above, we've seen the content of files *component_a.ads*, *component_a.adb* and *main_system.adb*. Now, let's discuss how to write the project file for Component_A (*component_a.gpr*), which will build the dynamic library for this component:

```

library project Component_A is

  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Create_Missing_Dirs use "True";
  for Library_Name use "component_a";
  for Library_Kind use "dynamic";
  for Library_Dir use "lib";

end Component_A;

```

The project is defined as a *library project* instead of *project*. This tells **GPRbuild** to build a library instead of an executable binary. We then specify the library name using the *Library_Name* attribute, which is required, so it must appear in a library project. The next two library-related attributes are optional, but important for our use-case. We use:

- *Library_Kind* to specify that we want to create a dynamic library — by default, this attribute is set to *static*;
- *Library_Dir* to specify the directory where the library is stored.

In the project file of our main system (*main_system.gpr*), we just need to reference the project of Component_A using a *with* clause and indicating the correct path to that project file:

```

with ".../component_a/component_a.gpr";

project Main_System is
  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Create_Missing_Dirs use "True";
  for Main use ("main_system.adb");
end Main_System;

```

GPRbuild takes care of selecting the correct settings to link the dynamic library created for Component_A with the main application (Main_System) and build an executable.

We can use the same strategy to create a Component_B and dynamically link to it in the Main_System. We just need to create the separate structure for this component — with the appropriate Ada packages and project file — and include it in the project file of the main system using a *with* clause:

```
with ".../component_a/component_a.gpr";
with ".../component_b/component_b.gpr";
...
...
```

Again, **GPRbuild** takes care of selecting the correct settings to link both dynamic libraries together with the main application.

You can find more details and special setting for library projects in the [GPRbuild documentation](#)¹⁰⁹.

In the GNAT toolchain

The GNAT toolchain includes a more advanced example focusing on how to load dynamic libraries at runtime. You can find it in the share/examples/gnat/plugins directory of the GNAT toolchain installation. As described in the README file from that directory, this example "comprises a main program which probes regularly for the existence of shared libraries in a known location. If such libraries are present, it uses them to implement features initially not present in the main program."

¹⁰⁹ https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug/gnat_project_manager.html#library-projects

PERFORMANCE CONSIDERATIONS

69.1 Overall expectations

All in all, there should not be significant performance differences between code written in Ada and code written in C, provided that they are semantically equivalent. Taking the current GNAT implementation and its GCC C counterpart for example, most of the code generation and optimization phases are shared between C and Ada — so there's not one compiler more efficient than the other. Furthermore, the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. They should be equivalent on average.

When comparing the performance of C and Ada code, differences might be observed. This usually comes from the fact that, while the two piece *appear* semantically equivalent, they happen to be actually quite different; C code semantics do not implicitly apply the same run-time checks that Ada does. This section will present common ways for improving Ada code performance.

69.2 Switches and optimizations

Clever use of compilation switches might optimize the performance of an application significantly. In this section, we'll briefly look into some of the switches available in the GNAT toolchain.

69.2.1 Optimizations levels

Optimization levels can be found in many compilers for multiple languages. On the lowest level, the GNAT compiler doesn't optimize the code at all, while at the higher levels, the compiler analyses the code and optimizes it by removing unnecessary operations and making the most use of the target processor's capabilities.

By being part of GCC, GNAT offers the same `-O` switches as GCC:

Switch	Description
<code>-O0</code>	No optimization: the generated code is completely unoptimized. This is the default optimization level.
<code>-O1</code>	Moderate optimization.
<code>-O2</code>	Full optimization.
<code>-O3</code>	Same optimization level as for <code>-O2</code> . In addition, further optimization strategies, such as aggressive automatic inlining and vectorization.

Note that the higher the level, the longer the compilation time. For fast compilation during development phase, unless you're working on benchmarking algorithms, using -O0 is probably a good idea.

In addition to the levels presented above, GNAT also has the -Os switch, which allows for optimizing code and data usage.

69.2.2 Inlining

As we've seen in the previous section, automatic inlining depends on the optimization level. The highest optimization level (-O3), for example, performs aggressive automatic inlining. This could mean that this level inlines too much rather than not enough. As a result, the cache may become an issue and the overall performance may be worse than the one we would achieve by compiling the same code with optimization level 2 (-O2). Therefore, the general recommendation is to not just select -O3 for the optimized version of an application, but instead compare it to the optimized version built with -O2.

In some cases, it's better to reduce the optimization level and perform manual inlining instead of automatic inlining. We do that by using the `Inline` aspect. Let's reuse an example from a previous chapter and inline the `Average` function:

[Ada]

Listing 1: float_arrays.ads

```
1 package Float_Arrays is
2
3     type Float_Array is array (Positive range <>) of Float;
4
5     function Average (Data : Float_Array) return Float
6         with Inline;
7
8 end Float_Arrays;
```

Listing 2: float_arrays.adb

```
1 package body Float_Arrays is
2
3     function Average (Data : Float_Array) return Float is
4         Total : Float := 0.0;
5     begin
6         for Value of Data loop
7             Total := Total + Value;
8         end loop;
9         return Total / Float (Data'Length);
10    end Average;
11
12 end Float_Arrays;
```

Listing 3: compute_average.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Float_Arrays; use Float_Arrays;
4
5 procedure Compute_Average is
6     Values      : constant Float_Array := (10.0, 11.0, 12.0, 13.0);
7     Average_Value : Float;
8 begin
9     Average_Value := Average (Values);
```

(continues on next page)

(continued from previous page)

```
10 Put_Line ("Average = " & Float'Image (Average_Value));
11 end Compute_Average;
```

Runtime output

```
Average = 1.15000E+01
```

When compiling this example, GNAT will inline Average in the Compute_Average procedure.

In order to effectively use this aspect, however, we need to set the optimization level to at least -O1 and use the -gnatn switch, which instructs the compiler to take the **Inline** aspect into account.

Note, however, that the **Inline** aspect is just a *recommendation* to the compiler. Sometimes, the compiler might not be able to follow this recommendation, so it won't inline the subprogram. In this case, we get a compilation warning from GNAT.

These are some examples of situations where the compiler might not be able to inline a subprogram:

- when the code is too large,
- when it's too complicated — for example, when it involves exception handling —, or
- when it contains tasks, etc.

In addition to the **Inline** aspect, we also have the **Inline_Always** aspect. In contrast to the former aspect, however, the **Inline_Always** aspect isn't primarily related to performance. Instead, it should be used when the functionality would be incorrect if inlining was not performed by the compiler. Examples of this are procedures that insert Assembly instructions that only make sense when the procedure is inlined, such as memory barriers.

Similar to the **Inline** aspect, there might be situations where a subprogram has the **Inline_Always** aspect, but the compiler is unable to inline it. In this case, we get a compilation error from GNAT.

69.3 Checks and assertions

69.3.1 Checks

Ada provides many runtime checks to ensure that the implementation is working as expected. For example, when accessing an array, we would like to make sure that we're not accessing a memory position that is not allocated for that array. This is achieved by an index check.

Another example of runtime check is the verification of valid ranges. For example, when adding two integer numbers, we would like to ensure that the result is still in the valid range — that the value is neither too large nor too small. This is achieved by an range check. Likewise, arithmetic operations shouldn't overflow or underflow. This is achieved by an overflow check.

Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots. For example, checking the index of an array in a sorting algorithm may significantly decrease its performance. In those cases, suppressing the check may be an option. We can achieve this suppression by using **pragma Suppress (Index_Check)**. For example:

[Ada]

```
procedure Sort (A : in out Integer_Array) is
  pragma Suppress (Index_Check);
begin
  -- (implementation removed...)
  null;
end Sort;
```

In case of overflow checks, we can use `pragma Suppress (Overflow_Check)` to suppress them:

```
function Some_Computation (A, B : Int32) return Int32 is
  pragma Suppress (Overflow_Check);
begin
  -- (implementation removed...)
  null;
end Sort;
```

We can also deactivate overflow checks for integer types using the `-gnato` switch when compiling a source-code file with GNAT. In this case, overflow checks in the whole file are deactivated.

It is also possible to suppress all checks at once using `pragma Suppress (All_Checks)`. In addition, GNAT offers a compilation switch called `-gnatp`, which has the same effect on the whole file.

Note, however, that this kind of suppression is just a recommendation to the compiler. There's no guarantee that the compiler will actually suppress any of the checks because the compiler may not be able to do so — typically because the hardware happens to do it. For example, if the machine traps on any access via address zero, requesting the removal of null access value checks in the generated code won't prevent the checks from happening.

It is important to differentiate between required and redundant checks. Let's consider the following example in C:

[C]

Listing 4: main.c

```
1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int a = 8, b = 0, res;
6
7     res = a / b;
8
9     // printing the result
10    printf("res = %d\n", res);
11
12    return 0;
13 }
```

Because C doesn't have language-defined checks, as soon as the application tries to divide a value by zero in `res = a / b`, it'll break — on Linux, for example, you may get the following error message by the operating system: `Floating point exception (core dumped)`. Therefore, we need to manually introduce a check for zero before this operation. For example:

[C]

Listing 5: main.c

```

1 #include <stdio.h>
2
3 int main(int argc, const char * argv[])
4 {
5     int a = 8, b = 0, res;
6
7     if (b != 0) {
8         res = a / b;
9
10        // printing the result
11        printf("res = %d\n", res);
12    }
13    else
14    {
15        // printing error message
16        printf("Error: cannot calculate value (division by zero)\n");
17    }
18
19    return 0;
20 }
```

Runtime output

```
Error: cannot calculate value (division by zero)
```

This is the corresponding code in Ada:

[Ada]

Listing 6: show_division_by_zero.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Division_By_Zero is
4     A : Integer := 8;
5     B : Integer := 0;
6     Res : Integer;
7 begin
8     Res := A / B;
9
10    Put_Line ("Res = " & Integer'Image (Res));
11 end Show_Division_By_Zero;
```

Build output

```
show_division_by_zero.adb:4:04: warning: "A" is not modified, could be declared constant [-gnatwk]
show_division_by_zero.adb:5:04: warning: "B" is not modified, could be declared constant [-gnatwk]
show_division_by_zero.adb:8:15: warning: division by zero [enabled by default]
show_division_by_zero.adb:8:15: warning: Constraint_Error will be raised at run-time [enabled by default]
```

Runtime output

```
raised CONSTRAINT_ERROR : show_division_by_zero.adb:8 divide by zero
```

Similar to the first version of the C code, we're not explicitly checking for a potential division by zero here. In Ada, however, this check is *automatically inserted* by the language itself.

When running the application above, an exception is raised when the application tries to divide the value in A by zero. We could introduce exception handling in our example, so that we get the same message as we did in the second version of the C code:

[Ada]

Listing 7: show_division_by_zero.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Show_Division_By_Zero is
4      A    : Integer := 8;
5      B    : Integer := 0;
6      Res : Integer;
7  begin
8      Res := A / B;
9
10     Put_Line ("Res = " & Integer'Image (Res));
11  exception
12      when Constraint_Error =>
13          Put_Line ("Error: cannot calculate value (division by zero)");
14      when others =>
15          null;
16  end Show_Division_By_Zero;
```

Build output

```
show_division_by_zero.adb:4:04: warning: "A" is not modified, could be declared constant \[-gnatwk\]
show_division_by_zero.adb:5:04: warning: "B" is not modified, could be declared constant \[-gnatwk\]
show_division_by_zero.adb:8:15: warning: division by zero [enabled by default]
show_division_by_zero.adb:8:15: warning: Constraint_Error will be raised at run time \[enabled by default\]
```

Runtime output

```
Error: cannot calculate value (division by zero)
```

This example demonstrates that the division check for `Res := A / B` is required and shouldn't be suppressed. In contrast, a check is redundant — and therefore not required — when we know that the condition that leads to a failure can never happen. In many cases, the compiler itself detects redundant checks and eliminates them (for higher optimization levels). Therefore, when improving the performance of your application, you should:

1. keep all checks active for most parts of the application;
2. identify the hot-spots of your application;
3. identify which checks haven't been eliminated by the optimizer on these hot-spots;
4. identify which of those checks are redundant;
5. only suppress those checks that are redundant, and keep the required ones.

69.3.2 Assertions

We've already discussed assertions in [this section of the SPARK chapter](#) (page 731). Assertions are user-defined checks that you can add to your code using the **pragma Assert**. For example:

[Ada]

```
function Some_Computation (A, B : Int32) return Int32 is
    Res : Int32;
begin
    -- (implementation removed...)

    pragma Assert (Res >= 0);

    return Res;
end Sort;
```

Assertions that are specified with **pragma Assert** are not enabled by default. You can enable them by setting the assertion policy to *check* — using **pragma Assertion_Policy** (*Check*) — or by using the `-gnata` switch when compiling with GNAT.

Similar to the checks discussed previously, assertions can generate significant overhead when used at hot-spots. Restricting those assertions to development (e.g. debug version) and turning them off on the release version may be an option. In this case, formal proof — as discussed in the [SPARK chapter](#) (page 725) — can help you. By formally proving that assertions will never fail at run-time, you can safely deactivate them.

69.4 Dynamic vs. static structures

Ada generally speaking provides more ways than C or C++ to write simple dynamic structures, that is to say structures that have constraints computed after variables. For example, it's quite typical to have initial values in record types:

[Ada]

```
type R is record
    F : Some_Field := Call_To_Some_Function;
end record;
```

However, the consequences of the above is that any declaration of a instance of this type without an explicit value for F will issue a call to `Call_To_Some_Function`. More subtle issue may arise with elaboration. For example, it's possible to write:

Listing 8: some_functions.ads

```
1 package Some_Functions is
2
3     function Some_Function_Call return Integer is (2);
4
5     function Some_Other_Function_Call return Integer is (10);
6
7 end Some_Functions;
```

Listing 9: values.ads

```
1 with Some_Functions; use Some_Functions;
2
3 package Values is
```

(continues on next page)

(continued from previous page)

```

4 A_Start : Integer := Some_Function_Call;
5 A_End   : Integer := Some_Other_Function_Call;
6 end Values;
```

Listing 10: arr_def.ads

```

1 with Values; use Values;
2
3 package Arr_Def is
4     type Arr is array (Integer range A_Start .. A_End) of Integer;
5 end Arr_Def;
```

It may indeed be appealing to be able to change the values of A_Start and A_End at startup so as to align a series of arrays dynamically. The consequence, however, is that these values will not be known statically, so any code that needs to access to boundaries of the array will need to read data from memory. While it's perfectly fine most of the time, there may be situations where performances are so critical that static values for array boundaries must be enforced.

Here's a last case which may also be surprising:

[Ada]

Listing 11: arr_def.ads

```

1 package Arr_Def is
2     type Arr is array (Integer range <>) of Integer;
3
4     type R (D1, D2 : Integer) is record
5         F1 : Arr (1 .. D1);
6         F2 : Arr (1 .. D2);
7     end record;
8 end Arr_Def;
```

In the code above, R contains two arrays, F1 and F2, respectively constrained by the discriminant D1 and D2. The consequence is, however, that to access F2, the run-time needs to know how large F1 is, which is dynamically constrained when creating an instance. Therefore, accessing to F2 requires a computation involving D1 which is slower than, let's say, two pointers in an C array that would point to two different arrays.

Generally speaking, when values are used in data structures, it's useful to always consider where they're coming from, and if their value is static (computed by the compiler) or dynamic (only known at run-time). There's nothing fundamentally wrong with dynamically constrained types, unless they appear in performance-critical pieces of the application.

69.5 Pointers vs. data copies

In the section about [pointers](#) (page 750), we mentioned that the Ada compiler will automatically pass parameters by reference when needed. Let's look into what "when needed" means. The fundamental point to understand is that the parameter types determine how the parameters are passed in and/or out. The parameter modes do not control how parameters are passed.

Specifically, the language standards specifies that scalar types are always passed by value, and that some other types are always passed by reference. It would not make sense to make a copy of a task when passing it as a parameter, for example. So parameters that can be passed reasonably by value will be, and those that must be passed by reference will be. That's the safest approach.

But the language also specifies that when the parameter is an array type or a record type, and the record/array components are all by-value types, then the compiler decides: it can pass the parameter using either mechanism. The critical case is when such a parameter is large, e.g., a large matrix. We don't want the compiler to pass it by value because that would entail a large copy, and indeed the compiler will not do so. But if the array or record parameter is small, say the same size as an address, then it doesn't matter how it is passed and by copy is just as fast as by reference. That's why the language gives the choice to the compiler. Although the language does not mandate that large parameters be passed by reference, any reasonable compiler will do the right thing.

The modes do have an effect, but not in determining how the parameters are passed. Their effect, for parameters passed by value, is to determine how many times the value is copied. For mode `in` and mode `out` there is just one copy. For mode `in out` there will be two copies, one in each direction.

Therefore, unlike C, you don't have to use access types in Ada to get better performance when passing arrays or records to subprograms. The compiler will almost certainly do the right thing for you.

Let's look at this example:

[C]

Listing 12: main.c

```

1 #include <stdio.h>
2
3 struct Data {
4     int prev, curr;
5 };
6
7 void update(struct Data *d,
8             int v)
9 {
10    d->prev = d->curr;
11    d->curr = v;
12 }
13
14 void display(const struct Data *d)
15 {
16    printf("Prev : %d\n", d->prev);
17    printf("Curr : %d\n", d->curr);
18 }
19
20 int main(int argc, const char * argv[])
21 {
22     struct Data D1 = { 0, 1 };
23
24     update (&D1, 3);
25     display (&D1);
26
27     return 0;
28 }
```

Runtime output

```
Prev : 1
Curr : 3
```

In this C code example, we're using pointers to pass `D1` as a reference to `update` and `display`. In contrast, the equivalent code in Ada simply uses the parameter modes to specify the data flow directions. The mechanisms used to pass the values do not appear in the source code.

[Ada]

Listing 13: update_record.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Update_Record is
4
5      type Data is record
6          Prev : Integer;
7          Curr : Integer;
8      end record;
9
10     procedure Update (D : in out Data;
11                      V :           Integer) is
12     begin
13         D.Prev := D.Curr;
14         D.Curr := V;
15     end Update;
16
17     procedure Display (D : Data) is
18     begin
19         Put_Line ("Prev: " & Integer'Image (D.Prev));
20         Put_Line ("Curr: " & Integer'Image (D.Curr));
21     end Display;
22
23     D1 : Data := (0, 1);
24
25 begin
26     Update (D1, 3);
27     Display (D1);
28 end Update_Record;
```

Runtime output

```
Prev: 1
Curr: 3
```

In the calls to `Update` and `Display`, `D1` is always passed by reference. Because no extra copy takes place, we get a performance that is equivalent to the C version. If we had used arrays in the example above, `D1` would have been passed by value as well:

[Ada]

Listing 14: update_array.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Update_Array is
4
5      type Data_State is (Prev, Curr);
6      type Data is array (Data_State) of Integer;
7
8      procedure Update (D : in out Data;
9                        V :           Integer) is
10     begin
11         D (Prev) := D (Curr);
12         D (Curr) := V;
13     end Update;
14
15     procedure Display (D : Data) is
16     begin
17         Put_Line ("Prev: " & Integer'Image (D (Prev)));
```

(continues on next page)

(continued from previous page)

```

18   Put_Line ("Curr: " & Integer'Image (D (Curr)));
19 end Display;
20
21 D1 : Data := (0, 1);
22
23 begin
24   Update (D1, 3);
25   Display (D1);
26 end Update_Array;

```

Runtime output

```

Prev: 1
Curr: 3

```

Again, no extra copy is performed in the calls to `Update` and `Display`, which gives us optimal performance when dealing with arrays and avoids the need to use access types to optimize the code.

69.5.1 Function returns

Previously, we've discussed the cost of passing complex records as arguments to subprograms. We've seen that we don't have to use explicit access type parameters to get better performance in Ada. In this section, we'll briefly discuss the cost of function returns.

In general, we can use either procedures or functions to initialize a data structure. Let's look at this example in C:

[C]

Listing 15: main.c

```

1 #include <stdio.h>
2
3 struct Data {
4     int prev, curr;
5 };
6
7 void init_data(struct Data *d)
8 {
9     d->prev = 0;
10    d->curr = 1;
11 }
12
13 struct Data get_init_data()
14 {
15     struct Data d = { 0, 1 };
16
17     return d;
18 }
19
20 int main(int argc, const char * argv[])
21 {
22     struct Data D1;
23
24     D1 = get_init_data();
25
26     init_data(&D1);
27

```

(continues on next page)

(continued from previous page)

```
28     return 0;
29 }
```

This code example contains two subprograms that initialize the Data structure:

- `init_data()`, which receives the data structure as a reference (using a pointer) and initializes it, and
- `get_init_data()`, which returns the initialized structure.

In C, we generally avoid implementing functions such as `get_init_data()` because of the extra copy that is needed for the function return.

This is the corresponding implementation in Ada:

[Ada]

Listing 16: init_record.adb

```
1  procedure Init_Record is
2
3      type Data is record
4          Prev : Integer;
5          Curr : Integer;
6      end record;
7
8      procedure Init (D : out Data) is
9      begin
10         D := (Prev => 0, Curr => 1);
11     end Init;
12
13      function Init return Data is
14          D : constant Data := (Prev => 0, Curr => 1);
15      begin
16         return D;
17     end Init;
18
19     D1 : Data;
20
21     pragma Unreferenced (D1);
22 begin
23     D1 := Init;
24
25     Init (D1);
26 end Init_Record;
```

Build output

```
init_record.adb:25:10: warning: pragma Unreferenced given for "D1" [enabled by
→default]
```

In this example, we have two versions of `Init`: one using a procedural form, and the other one using a functional form. Note that, because of Ada's support for subprogram overloading, we can use the same name for both subprograms.

The issue is that assignment of a function result entails a copy, just as if we assigned one variable to another. For example, when assigning a function result to a constant, the function result is copied into the memory for the constant. That's what is happening in the above examples for the initialized variables.

Therefore, in terms of performance, the same recommendations apply: for large types we should avoid writing functions like the `Init` function above. Instead, we should use the procedural form of `Init`. The reason is that the compiler necessarily generates a copy for

the `Init` function, while the `Init` procedure uses a reference for the output parameter, so that the actual record initialization is performed in place in the caller's argument.

An exception to this is when we use functions returning values of limited types, which by definition do not allow assignment. Here, to avoid allowing something that would otherwise look suspiciously like an assignment, the compiler generates the function body so that it builds the result directly into the object being assigned. No copy takes place.

We could, for example, rewrite the example above using limited types:

[Ada]

Listing 17: init_limited_record.adb

```

1  procedure Init_Limited_Record is
2
3      type Data is limited record
4          Prev : Integer;
5          Curr : Integer;
6      end record;
7
8      function Init return Data is
9      begin
10         return D : Data do
11             D.Prev := 0;
12             D.Curr := 1;
13         end return;
14     end Init;
15
16     D1 : Data := Init;
17
18     pragma Unreferenced (D1);
19 begin
20     null;
21 end Init_Limited_Record;
```

In this example, `D1 := Init;` has the same cost as the call to the procedural form — `Init (D1);` — that we've seen in the previous example. This is because the assignment is done in place.

Note that limited types require the use of the extended return statements (`return ... do` ... `end return`) in function implementations. Also note that, because the `Data` type is limited, we can only use the `Init` function in the declaration of `D1`; a statement in the code such as `D1 := Init;` is therefore forbidden.

ARGUMENTATION AND BUSINESS PERSPECTIVES

The technical benefits of a migration from C to Ada are usually relatively straightforward to demonstrate. Hopefully, this course provides a good basis for it. However, when faced with an actual business decision to make, additional considerations need to be taken into account, such as return on investment, perennity of the solution, tool support, etc. This section will cover a number of usual questions and provide elements of answers.

70.1 What's the expected ROI of a C to Ada transition?

Switching from one technology to another is a cost, may that be in terms of training, transition of the existing environment or acquisition of new tools. This investment needs to be matched with an expected return on investment, or ROI, to be consistent. Of course, it's incredibly difficult to provide a firm answer to how much money can be saved by transitioning, as this is highly dependent on specific project objectives and constraints. We're going to provide qualitative and quantitative arguments here, from the perspective of a project that has to reach a relatively high level of integrity, that is to say a system where the occurrence of a software failure is a relatively costly event.

From a qualitative standpoint, there are various times in the software development life cycle where defects can be found:

1. on the developer's desk
2. during component testing
3. during integration testing
4. after deployment
5. during maintenance

Numbers from studies vary greatly on the relative costs of defects found at each of these phases, but there's a clear ordering between them. For example, a defect found while developing is orders of magnitude less expensive to fix than a defect found e.g. at integration time, which may involve costly debugging sessions and slow down the entire system acceptance. The whole purpose of Ada and SPARK is to push defect detection to the developer's desk as much as possible; at least for all of these defects that can be identified at that level. While the strict act of writing software may be taking more effort because of all of the additional safeguards, this should have a significant and positive impact down the line and help to control costs overall. The exact value this may translate into is highly business dependent.

From a quantitative standpoint, two studies have been done almost 25 years apart and provide similar insights:

- Rational Software in 1995 found that the cost of developing software in Ada was overall half as much as the cost of developing software in C.

- VDC ran a study in 2018, finding that the cost savings of developing with Ada over C ranged from 6% to 38% in savings.

From a qualitative standpoint, in particular with regards to Ada and C from a formal proof perspective, an interesting presentation was made in 2017 by two researchers. They tried to apply formal proof on the same piece of code, developed in Ada/SPARK on one end and C/Frama-C on the other. Their results indicate that the Ada/SPARK technology is indeed more conducive to formal proof methodologies.

Although all of these studies have their own biases, they provide a good idea of what to expect in terms of savings once the initial investment in switching to Ada is made. This is assuming everything else is equal, in particular that the level of integrity is the same. In many situations, the migration to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will rise (it's more expensive to develop better software) and Ada is viewed as a means to mitigate this rise in development costs.

That being said, the point of this argument is not to say that it's not possible to write very safe and secure software with languages different than Ada. With the right expertise, the right processes and the right tools, it's done every day. The point is that Ada overall reduces the level of processes, expertise and tools necessary and will allow to reach the same target at a lower cost.

70.2 Who is using Ada today?

Ada was initially born as a DoD project, and thus got its initial customer base in aerospace and defence (A&D). At the time these lines are written and from the perspective of AdaCore, A&D is still the largest consumer of Ada today and covers about 70% of the market. This creates a consistent and long lasting set of established users as these projects last often for decades, using the same codebase migrating from platform to platform.

More recently however, there has been an emerging interest for Ada in new communities of users such as automotive, medical device, industrial automation and overall cyber-security. This can probably be explained by a rise of safety, reliability and cyber-security requirements. The market is moving relatively rapidly today and we're anticipating an increase of the Ada footprint in these domains, while still remaining a technology of choice for the development of mission critical software.

70.3 What is the future of the Ada technology?

The first piece of the answer lies in the user base of the Ada language, as seen in the previous question. Projects using Ada in the aerospace and defence domain maintain source code over decades, providing healthy funding foundation for Ada-based technologies.

AdaCore being the author of this course, it's difficult for us to be fair in our description of other Ada compilation technologies. We will leave to the readers the responsibility of forging their own opinion. If they present a credible alternative to the GNAT compiler, then this whole section can be considered as void.

Assuming GNAT is the only option available, and acknowledging that this is an argument that we're hearing from a number of Ada adopters, let's discuss the "sole source" issue.

First of all, it's worth noting that industries are using a lot of software that is provided by only one source, so while non-ideal, these situations are also quite common.

In the case of the GNAT compiler however, while AdaCore is the main maintainer, this maintenance is done as part of an open-source community. This means that nothing prevents

a third party to start selling a competing set of products based on the same compiler, provided that it too adopts the open-source approach. Our job is to be more cost-effective than the alternative, and indeed for the vast part this has prevented a competing offering to emerge. However, should AdaCore disappear or switch focus, Ada users would not be prevented from carrying on using its software (there is no lock) and a third party could take over maintenance. This is not a theoretical case, this has been done in the past either by companies looking at supporting their own version of GNAT, vendors occupying a specific niche that was left uncovered , or hobbyists developing their own builds.

With that in mind, it's clear that the "sole source" provider issue is a circumstantial — nothing is preventing other vendors from emerging if the conditions are met.

70.4 Is the Ada toolset complete?

A language by itself is of little use for the development of safety-critical software. Instead, a complete toolset is needed to accompany the development process, in particular tools for edition, testing, static analysis, etc.

AdaCore provides a number of these tools either in through its core or add-on package. These include (as of 2019):

- An IDE (GNAT Studio)
- An Eclipse plug-in (GNATbench)
- A debugger (GDB)
- A testing tool (GNATtest)
- A structural code coverage tool (GNATcoverage)
- A metric computation tool (GNATmetric)
- A coding standard checker (GNATcheck)
- Static analysis tools (CodePeer, SPARK Pro)
- A Simulink code generator (QGen)
- An Ada parser to develop custom tools (libadalang)

Ada is, however, an internationally standardized language, and many companies are providing third party solutions to complete the toolset. Overall, the language can be and is used with tools on par with their equivalent C counterparts.

70.5 Where can I find Ada or SPARK developers?

A common question from teams on the verge of selecting Ada and SPARK is how to manage the developer team growth and turnover. While Ada and SPARK are taught by a growing number of universities worldwide, it may still be challenging to hire new staff with prior Ada experience.

Fortunately, Ada's base semantics are very close to those of C/C++, so that a good embedded software developer should be able to learn it relatively easily. This course is definitely a resource available to get started. Online training material is also available, together with on-site in person training.

In general, getting an engineer operational in Ada and SPARK shouldn't take more than a few weeks worth of time.

70.6 How to introduce Ada and SPARK in an existing code base?

The most common scenario when introducing Ada and SPARK to a project or a team is to do it within a pre-existing C codebase, which can already spread over hundreds of thousands if not millions lines of code. Re-writing this software to Ada or SPARK is of course not practical and counterproductive.

Most teams select either a small piece of existing code which deserves particular attention, or new modules to develop, and concentrate on this. Developing this module or part of the application will also help in developing the coding patterns to be used for the particular project and company. This typically concentrates an effort of a few people on a few thousands lines of code. The resulting code can be linked to the rest of the C application. From there, the newly established practices and their benefit can slowly spread through the rest of the environment.

Establishing this initial core in Ada and SPARK is critical, and while learning the language isn't a particularly difficult task, applying it to its full capacity may require some expertise. One possibility to accelerate this initial process is to use AdaCore mentorship services.

CHAPTER
SEVENTYONE

CONCLUSION

Although Ada's syntax might seem peculiar to C developers at first glance, it was designed to increase readability and maintainability, rather than making it faster to write in a condensed manner — as it is often the case in C.

Especially in the embedded domain, C developers are used to working at a very low level, which includes mathematical operations on pointers, complex bit shifts, and logical bitwise operations. C is well designed for such operations because it was designed to replace Assembly language for faster, more efficient programming.

Ada can be used to describe high level semantics and architectures. The beauty of the language, however, is that it can be used all the way down to the lowest levels of the development, including embedded Assembly code or bit-level data management. However, although Ada supports bitwise operations such as masks and shifts, they should be relatively rarely needed. When translating C code to Ada, it's good practice to consider alternatives. In a lot of cases, these operations are used to insert several pieces of data into a larger structure. In Ada, this can be done by describing the structure layout at the type level through representation clauses, and then accessing this structure as any other. For example, we can interpret an arbitrary data type as a bit-field and perform low-level operations on it.

Because Ada is a strongly typed language, it doesn't define any implicit type conversions like C. If we try to compile Ada code that contains type mismatches, we'll get a compilation error. Because the compiler prevents mixing variables of different types without explicit type conversion, we can't accidentally end up in a situation where we assume something will happen implicitly when, in fact, our assumption is incorrect. In this sense, Ada's type system encourages programmers to think about data at a high level of abstraction. Ada supports overlays and unchecked conversions as a way of converting between unrelated data type, which are typically used for interfacing with low-level elements such as registers.

In Ada, arrays aren't interchangeable with operations on pointers like in C. Also, array types are considered first-class citizens and have dedicated semantics such as the availability of the array's boundaries at run-time. Therefore, unhandled array overflows are impossible unless checks are suppressed. Any discrete type can serve as an array index, and we can specify both the starting and ending bounds. In addition, Ada offers high-level operations for copying, slicing, and assigning values to arrays.

Although Ada supports pointers, most situations that would require a pointer in C do not in Ada. In the vast majority of the cases, indirect memory management can be hidden from the developer and thus prevent many potential errors. In C, pointers are typically used to pass references to subprograms, for example. In contrast, Ada parameter modes indicate the flow of information to the reader, leaving the means of passing that information to the compiler.

When translating pointers from C code to Ada, we need to assess whether they are needed in the first place. Ada pointers (access types) should only be used with complex structures that cannot be allocated at run-time. There are many situations that would require a pointer in C, but do not in Ada. For example, arrays — even when dynamically allocated —, results of functions, passing of large structures as parameters, access to registers, etc.

Because of the absence of namespaces, global names in C tend to be very long. Also, because of the absence of overloading, they can even encode type names in their name. In Ada, a package is a namespace. Also, we can use the private part of a package to declare private types and private subprograms. In fact, private types are useful for preventing the users of those types from depending on the implementation details. Another use-case is the prevention of package users from accessing the package state/data arbitrarily.

Ada has a dedicated set of features for interfacing with other languages, so we can easily interface with our existing C code before translating it to Ada. Also, GNAT includes automatic binding generators. Therefore, instead of re-writing the entire C code upfront, which isn't practical or cost-effective, we can selectively translate modules from C to Ada.

When it comes to implementing concurrency and real time, Ada offers several options. Ada provides high level constructs such as tasks and protected objects to express concurrency and synchronization, which can be used when running on top of an operating system such as Linux. On more constrained systems, such as bare metal or some real-time operating systems, a subset of the Ada tasking capabilities — known as the Ravenscar and Jorvik profiles — is available. Though restricted, this subset also has nice properties, in particular the absence of deadlock, the absence of priority inversion, schedulability and very small footprint. On bare metal systems, this also essentially means that Ada comes with its own real-time kernel. The advantage of using the full Ada tasking model or the restricted profiles is to enhance portability.

Ada includes many features typically used for embedded programming:

- Built-in support for handling interrupts, so we can process interrupts by attaching a handler — as a protected procedure — to it.
- Built-in support for handling both volatile and atomic data.
- Support for register overlays, which we can use to create a structure that facilitates manipulating bits from registers.
- Support for creating data streams for serialization of arbitrary information and transmission over a communication channel, such as a serial port.
- Built-in support for fixed-point arithmetic, which is an option when our target device doesn't have a floating-point unit or the result of calculations needs to be bit-exact.

Also, Ada compilers such as GNAT have built-in support for directly mixing Ada and Assembly code.

Ada also supports contracts, which can be associated with types and variables to refine values and define valid and invalid values. The most common kind of contract is a *range constraint* — using the **range** reserved word. Ada also supports contract-based programming in the form of preconditions and postconditions. One typical benefit of contract-based programming is the removal of defensive code in subprogram implementations.

It is common to see embedded software being used in a variety of configurations that require small changes to the code for each instance. In C, variability is usually achieved through macros and function pointers, the former being tied to static variability and the latter to dynamic variability. Ada offers many alternatives for both techniques, which aim at structuring possible variations of the software. Examples of static variability in Ada are: genericity, simple derivation, configuration pragma files, and configuration packages. Examples of dynamic variability in Ada are: records with discriminants, variant records — which may include the use of unions —, object orientation, pointers to subprograms, and design by components using dynamic libraries.

There shouldn't be significant performance differences between code written in Ada and code written in C — provided that they are semantically equivalent. One reason is that the two languages are fairly similar in the way they implement imperative semantics, in particular with regards to memory management or control flow. Therefore, they should be equivalent on average. However, when a piece of code in Ada is significantly slower than its counterpart in C, this usually comes from the fact that, while the two pieces of code appear

to be semantically equivalent, they happen to be actually quite different. Fortunately, there are strategies that we can use to improve the performance and make it equivalent to the C version. These are some examples:

- Clever use of compilation switches, which might optimize the performance of an application significantly.
- Suppression of checks at specific parts of the implementation.
 - Although runtime checks are very useful and should be used as much as possible, they can also increase the overhead of implementations at certain hot-spots.
- Restriction of assertions to development code.
 - For example, we may use assertions in the debug version of the code and turn them off in the release version.
 - Also, we may use formal proof to decide which assertions we turn off in the release version. By formally proving that assertions will never fail at run-time, we can safely deactivate them.

Formal proof — a form of static analysis — can give strong guarantees about checks, for all possible conditions and all possible inputs. It verifies conditions prior to execution, even prior to compilation, so we can remove bugs earlier in the development phase. This is far less expensive than doing so later because the cost to fix bugs increases exponentially over the phases of the project life cycle, especially after deployment. Preventing bug introduction into the deployed system is the least expensive approach of all.

Formal analysis for proof can be achieved through the SPARK subset of the Ada language combined with the **gnatprove** verification tool. SPARK is a subset encompassing most of the Ada language, except for features that preclude proof.

In Ada, several common programming errors that are not already detected at compile-time are detected instead at run-time, triggering *exceptions* that interrupt the normal flow of execution. However, we may be able to prove that the language-defined checks won't raise exceptions at run-time. This is known as proving *Absence of Run-Time Errors*. Successful proof of these checks is highly significant in itself. One of the major resulting benefits is that we can deploy the final executable with checks disabled.

In many situations, the migration of C code to Ada is justified by an increase in terms of integrity expectations, in which case it's expected that development costs will raise. However, Ada is a more expressive, powerful language, designed to reduce errors earlier in the life-cycle, thus reducing costs. Therefore, Ada makes it possible to write very safe and secure software at a lower cost than languages such as C.

APPENDIX A: HANDS-ON OBJECT-ORIENTED PROGRAMMING

The goal of this appendix is to present a hands-on view on how to translate a system from C to Ada and improve it with object-oriented programming.

72.1 System Overview

Let's start with an overview of a simple system that we'll implement and use below. The main system is called AB and it combines two systems A and B. System AB is not supposed to do anything useful. However, it can serve as a good model for the hands-on we're about to start.

This is a list of requirements for the individual systems A and B, and the combined system AB:

- System A:
 - The system can be activated and deactivated.
 - * During activation, the system's values are reset.
 - Its current value (in floating-point) can be retrieved.
 - * This value is the average of the two internal floating-point values.
 - Its current state (activated or deactivated) can be retrieved.
- System B:
 - The system can be activated and deactivated.
 - * During activation, the system's value is reset.
 - Its current value (in floating-point) can be retrieved.
 - Its current state (activated or deactivated) can be retrieved.
- System AB
 - The system contains an instance of system A and an instance of system B.
 - The system can be activated and deactivated.
 - * System AB activates both systems A and B during its own activation.
 - * System AB deactivates both systems A and B during its own deactivation.
 - Its current value (in floating-point) can be retrieved.
 - * This value is the average of the current values of systems A and B.
 - Its current state (activated or deactivated) can be retrieved.
 - * AB is only considered activated when both systems A and B are activated.

- The system's health can be checked.
 - * This check consists in calculating the absolute difference D between the current values of systems A and B and checking whether D is below a threshold of 0.1.

The source-code in the following section contains an implementation of these requirements.

72.2 Non Object-Oriented Approach

In this section, we look into implementations (in both C and Ada) of system AB that don't make use of object-oriented programming.

72.2.1 Starting point in C

Let's start with an implementation in C for the system described above:

[C]

Listing 1: system_a.h

```

1 typedef struct {
2     float val[2];
3     int active;
4 } A;
5
6 void A_activate (A *a);
7
8 int A_is_active (A *a);
9
10 float A_value (A *a);
11
12 void A_deactivate (A *a);

```

Listing 2: system_a.c

```

1 #include "system_a.h"
2
3 void A_activate (A *a)
4 {
5     int i;
6
7     for (i = 0; i < 2; i++)
8     {
9         a->val[i] = 0.0;
10    }
11    a->active = 1;
12 }
13
14 int A_is_active (A *a)
15 {
16     return a->active == 1;
17 }
18
19 float A_value (A *a)
20 {
21     return (a->val[0] + a->val[1]) / 2.0;
22 }

```

(continues on next page)

(continued from previous page)

```

23 void A_deactivate (A *a)
24 {
25     a->active = 0;
26 }
27

```

Listing 3: system_b.h

```

1  typedef struct {
2      float val;
3      int active;
4 } B;
5
6 void B_activate (B *b);
7
8 int B_is_active (B *b);
9
10 float B_value (B *b);
11
12 void B_deactivate (B *b);

```

Listing 4: system_b.c

```

1 #include "system_b.h"
2
3 void B_activate (B *b)
4 {
5     b->val = 0.0;
6     b->active = 1;
7 }
8
9 int B_is_active (B *b)
10 {
11     return b->active == 1;
12 }
13
14 float B_value (B *b)
15 {
16     return b->val;
17 }
18
19 void B_deactivate (B *b)
20 {
21     b->active = 0;
22 }

```

Listing 5: system_ab.h

```

1 #include "system_a.h"
2 #include "system_b.h"
3
4 typedef struct {
5     A a;
6     B b;
7 } AB;
8
9 void AB_activate (AB *ab);
10
11 int AB_is_active (AB *ab);
12

```

(continues on next page)

(continued from previous page)

```

13 float AB_value (AB *ab);
14
15 int AB_check (AB *ab);
16
17 void AB_deactivate (AB *ab);

```

Listing 6: system_ab.c

```

1 #include <math.h>
2 #include "system_ab.h"
3
4 void AB_activate (AB *ab)
5 {
6     A_activate (&ab->a);
7     B_activate (&ab->b);
8 }
9
10 int AB_is_active (AB *ab)
11 {
12     return A_is_active(&ab->a) && B_is_active(&ab->b);
13 }
14
15 float AB_value (AB *ab)
16 {
17     return (A_value (&ab->a) + B_value (&ab->b)) / 2;
18 }
19
20 int AB_check (AB *ab)
21 {
22     const float threshold = 0.1;
23
24     return fabs (A_value (&ab->a) - B_value (&ab->b)) < threshold;
25 }
26
27 void AB_deactivate (AB *ab)
28 {
29     A_deactivate (&ab->a);
30     B_deactivate (&ab->b);
31 }

```

Listing 7: main.c

```

1 #include <stdio.h>
2 #include "system_ab.h"
3
4 void display_active (AB *ab)
5 {
6     if (AB_is_active (ab))
7         printf ("System AB is active.\n");
8     else
9         printf ("System AB is not active.\n");
10 }
11
12 void display_check (AB *ab)
13 {
14     if (AB_check (ab))
15         printf ("System AB check: PASSED.\n");
16     else
17         printf ("System AB check: FAILED.\n");
18 }

```

(continues on next page)

(continued from previous page)

```

19 int main()
20 {
21     AB s;
22
23     printf ("Activating system AB...\n");
24     AB_activate (&s);
25
26     display_active (&s);
27     display_check (&s);
28
29     printf ("Deactivating system AB...\n");
30     AB_deactivate (&s);
31
32     display_active (&s);
33 }

```

Runtime output

```

Activating system AB...
System AB is active.
System AB check: PASSED.
Deactivating system AB...
System AB is not active.

```

Here, each system is implemented in a separate set of header and source-code files. For example, the API of system AB is in `system_ab.h` and its implementation in `system_ab.c`.

In the main application, we instantiate system AB and activate it. Then, we proceed to display the activation state and the result of the system's health check. Finally, we deactivate the system and display the activation state again.

72.2.2 Initial translation to Ada

The direct implementation in Ada is:

[Ada]

Listing 8: `system_a.ads`

```

1 package System_A is
2
3     type Val_Array is array (Positive range <>) of Float;
4
5     type A is record
6         Val      : Val_Array (1 .. 2);
7         Active   : Boolean;
8     end record;
9
10    procedure A_Activate (E : in out A);
11
12    function A_Is_Active (E : A) return Boolean;
13
14    function A_Value (E : A) return Float;
15
16    procedure A_Deactivate (E : in out A);
17
18 end System_A;

```

Listing 9: system_a.adb

```
1 package body System_A is
2
3     procedure A_Activate (E : in out A) is
4 begin
5         E.Val := (others => 0.0);
6         E.Active := True;
7     end A_Activate;
8
9     function A_Is_Active (E : A) return Boolean is
10 begin
11     return E.Active;
12 end A_Is_Active;
13
14     function A_Value (E : A) return Float is
15 begin
16     return (E.Val (1) + E.Val (2)) / 2.0;
17 end A_Value;
18
19     procedure A_Deactivate (E : in out A) is
20 begin
21     E.Active := False;
22 end A_Deactivate;
23
24 end System_A;
```

Listing 10: system_b.ads

```
1 package System_B is
2
3     type B is record
4         Val : Float;
5         Active : Boolean;
6     end record;
7
8     procedure B_Activate (E : in out B);
9
10    function B_Is_Active (E : B) return Boolean;
11
12    function B_Value (E : B) return Float;
13
14    procedure B_Deactivate (E : in out B);
15
16 end System_B;
```

Listing 11: system_b.adb

```
1 package body System_B is
2
3     procedure B_Activate (E : in out B) is
4 begin
5         E.Val := 0.0;
6         E.Active := True;
7     end B_Activate;
8
9     function B_Is_Active (E : B) return Boolean is
10 begin
11     return E.Active;
12 end B_Is_Active;
```

(continues on next page)

(continued from previous page)

```

14  function B_Value (E : B) return Float is
15    begin
16      return E.Val;
17    end B_Value;
18
19  procedure B_Deactivate (E : in out B) is
20    begin
21      E.Active := False;
22    end B_Deactivate;
23
24 end System_B;

```

Listing 12: system_ab.ads

```

1  with System_A; use System_A;
2  with System_B; use System_B;
3
4  package System_AB is
5
6    type AB is record
7      SA : A;
8      SB : B;
9    end record;
10
11   procedure AB_Activate (E : in out AB);
12
13   function AB_Is_Active (E : AB) return Boolean;
14
15   function AB_Value (E : AB) return Float;
16
17   function AB_Check (E : AB) return Boolean;
18
19   procedure AB_Deactivate (E : in out AB);
20
21 end System_AB;

```

Listing 13: system_ab.adb

```

1  package body System_AB is
2
3    procedure AB_Activate (E : in out AB) is
4      begin
5        A_Activate (E.SA);
6        B_Activate (E.SB);
7      end AB_Activate;
8
9    function AB_Is_Active (E : AB) return Boolean is
10      begin
11        return A_Is_Active (E.SA) and B_Is_Active (E.SB);
12      end AB_Is_Active;
13
14    function AB_Value (E : AB) return Float is
15      begin
16        return (A_Value (E.SA) + B_Value (E.SB)) / 2.0;
17      end AB_Value;
18
19    function AB_Check (E : AB) return Boolean is
20      Threshold : constant := 0.1;
21      begin
22        return abs (A_Value (E.SA) - B_Value (E.SB)) < Threshold;

```

(continues on next page)

(continued from previous page)

```
23 end AB_Check;  
24  
25 procedure AB_Deactivate (E : in out AB) is  
26 begin  
27     A_Deactivate (E.SA);  
28     B_Deactivate (E.SB);  
29 end AB_Deactivate;  
30  
31 end System_AB;
```

Listing 14: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;  
2  
3 with System_AB; use System_AB;  
4  
5 procedure Main is  
6  
7 procedure Display_Active (E : AB) is  
8 begin  
9     if AB_Is_Active (E) then  
10         Put_Line ("System AB is active");  
11     else  
12         Put_Line ("System AB is not active");  
13     end if;  
14 end Display_Active;  
15  
16 procedure Display_Check (E : AB) is  
17 begin  
18     if AB_Check (E) then  
19         Put_Line ("System AB check: PASSED");  
20     else  
21         Put_Line ("System AB check: FAILED");  
22     end if;  
23 end Display_Check;  
24  
25 S : AB;  
26 begin  
27     Put_Line ("Activating system AB...");  
28     AB_Activate (S);  
29  
30     Display_Active (S);  
31     Display_Check (S);  
32  
33     Put_Line ("Deactivating system AB...");  
34     AB_Deactivate (S);  
35  
36     Display_Active (S);  
end Main;
```

Runtime output

```
Activating system AB...  
System AB is active  
System AB check: PASSED  
Deactivating system AB...  
System AB is not active
```

As you can see, this is a direct translation that doesn't change much of the structure of the original C code. Here, the goal was to simply translate the system from one language to another and make sure that the behavior remains the same.

72.2.3 Improved Ada implementation

By analyzing this direct implementation, we may notice the following points:

- Packages `System_A`, `System_B` and `System_AB` are used to describe aspects of the same system. Instead of having three distinct packages, we could group them as child packages of a common parent package — let's call it `Simple`, since this system is supposed to be simple. This approach has the advantage of allowing us to later use the parent package to implement functionality that is common for all parts of the system.
- Since we have subprograms that operate on types `A`, `B` and `AB`, we should avoid exposing the record components by moving the type declarations to the private part of the corresponding packages.
- Since Ada supports subprogram overloading — as discussed in [this section from chapter 2](#) (page 682) —, we don't need to have different names for subprograms with similar functionality. For example, instead of having `A_Is_Active` and `B_Is_Active`, we can simply name these functions `Is_Active` for both types `A` and `B`.
- Some of the functions — such as `A_Is_Active` and `A_Value` — are very simple, so we could simplify them with expression functions.

This is an update to the implementation that addresses all the points above:

[Ada]

Listing 15: `simple.ads`

```

1 package Simple
2   with Pure
3 is
4 end Simple;

```

Listing 16: `simple-system_a.ads`

```

1 package Simple.System_A is
2
3   type A is private;
4
5   procedure Activate (E : in out A);
6
7   function Is_Active (E : A) return Boolean;
8
9   function Value (E : A) return Float;
10
11  procedure Finalize (E : in out A);
12
13 private
14
15   type Val_Array is array (Positive range <>) of Float;
16
17   type A is record
18     Val : Val_Array (1 .. 2);
19     Active : Boolean;
20   end record;
21
22 end Simple.System_A;

```

Listing 17: `simple-system_a.adb`

```

1 package body Simple.System_A is
2

```

(continues on next page)

(continued from previous page)

```

3  procedure Activate (E : in out A) is
4  begin
5      E.Val    := (others => 0.0);
6      E.Active := True;
7  end Activate;
8
9  function Is_Active (E : A) return Boolean is
10     (E.Active);
11
12 function Value (E : A) return Float is
13 begin
14     return (E.Val (1) + E.Val (2)) / 2.0;
15 end Value;
16
17 procedure Finalize (E : in out A) is
18 begin
19     E.Active := False;
20 end Finalize;
21
22 end Simple.System_A;

```

Listing 18: simple-system_b.ads

```

1 package Simple.System_B is
2
3     type B is private;
4
5     procedure Activate (E : in out B);
6
7     function Is_Active (E : B) return Boolean;
8
9     function Value (E : B) return Float;
10
11    procedure Finalize (E : in out B);
12
13 private
14
15     type B is record
16         Val    : Float;
17         Active : Boolean;
18     end record;
19
20 end Simple.System_B;

```

Listing 19: simple-system_b.adb

```

1 package body Simple.System_B is
2
3     procedure Activate (E : in out B) is
4     begin
5         E.Val    := 0.0;
6         E.Active := True;
7     end Activate;
8
9     function Is_Active (E : B) return Boolean is
10    begin
11        return E.Active;
12    end Is_Active;
13
14     function Value (E : B) return Float is

```

(continues on next page)

(continued from previous page)

```

15   (E.Val);
16
17  procedure Finalize (E : in out B) is
18  begin
19    E.Active := False;
20  end Finalize;
21
22 end Simple.System_B;

```

Listing 20: simple-system_ab.ads

```

1 with Simple.System_A; use Simple.System_A;
2 with Simple.System_B; use Simple.System_B;
3
4 package Simple.System_AB is
5
6   type AB is private;
7
8   procedure Activate (E : in out AB);
9
10  function Is_Active (E : AB) return Boolean;
11
12  function Value (E : AB) return Float;
13
14  function Check (E : AB) return Boolean;
15
16  procedure Finalize (E : in out AB);
17
18 private
19
20  type AB is record
21    SA : A;
22    SB : B;
23  end record;
24
25 end Simple.System_AB;

```

Listing 21: simple-system_ab.adb

```

1 package body Simple.System_AB is
2
3   procedure Activate (E : in out AB) is
4   begin
5     Activate (E.SA);
6     Activate (E.SB);
7   end Activate;
8
9   function Is_Active (E : AB) return Boolean is
10    (Is_Active (E.SA) and Is_Active (E.SB));
11
12   function Value (E : AB) return Float is
13    ((Value (E.SA) + Value (E.SB)) / 2.0);
14
15   function Check (E : AB) return Boolean is
16    Threshold : constant := 0.1;
17   begin
18     return abs (Value (E.SA) - Value (E.SB)) < Threshold;
19   end Check;
20
21   procedure Finalize (E : in out AB) is

```

(continues on next page)

(continued from previous page)

```
22 begin
23     Finalize (E.SA);
24     Finalize (E.SB);
25 end Finalize;
26
27 end Simple.System_AB;
```

Listing 22: main.adb

```
1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Simple.System_AB; use Simple.System_AB;
4
5  procedure Main is
6
7      procedure Display_Active (E : AB) is
8          begin
9              if Is_Active (E) then
10                  Put_Line ("System AB is active");
11              else
12                  Put_Line ("System AB is not active");
13              end if;
14          end Display_Active;
15
16      procedure Display_Check (E : AB) is
17          begin
18              if Check (E) then
19                  Put_Line ("System AB check: PASSED");
20              else
21                  Put_Line ("System AB check: FAILED");
22              end if;
23          end Display_Check;
24
25      S : AB;
26      begin
27          Put_Line ("Activating system AB...");
28          Activate (S);
29
30          Display_Active (S);
31          Display_Check (S);
32
33          Put_Line ("Deactivating system AB...");
34          Finalize (S);
35
36          Display_Active (S);
37      end Main;
```

Runtime output

```
Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active
```

72.3 First Object-Oriented Approach

Until now, we haven't used any of the object-oriented programming features of the Ada language. So we can start by analyzing the API of systems A and B and deciding how to best abstract some of its elements using object-oriented programming.

72.3.1 Interfaces

The first thing we may notice is that we actually have two distinct sets of APIs there:

- one API for activating and deactivating the system.
- one API for retrieving the value of the system.

We can use this distinction to declare two interface types:

- Activation_IF for the Activate and Deactivate procedures and the Is_Active function;
- Value_Retrieval_IF for the Value function.

This is how the declaration could look like:

```
type Activation_IF is interface;

procedure Activate (E : in out Activation_IF) is abstract;
function Is_Active (E : Activation_IF) return Boolean is abstract;
procedure Deactivate (E : in out Activation_IF) is abstract;

type Value_Retrieval_IF is interface;

function Value (E : Value_Retrieval_IF) return Float is abstract;
```

Note that, because we are declaring interface types, all operations on those types must be abstract or, in the case of procedures, they can also be declared **null**. For example, we could change the declaration of the procedures above to this:

```
procedure Activate (E : in out Activation_IF) is null;
procedure Deactivate (E : in out Activation_IF) is null;
```

When an operation is declared abstract, we must override it for the type that derives from the interface. When a procedure is declared **null**, it acts as a do-nothing default. In this case, overriding the operation is optional for the type that derives from this interface.

72.3.2 Base type

Since the original system needs both interfaces we've just described, we have to declare another type that combines those interfaces. We can do this by declaring the interface type Sys_Base, which serves as the base type for systems A and B. This is the declaration:

```
type Sys_Base is interface and Activation_IF and Value_Retrieval_IF;
```

Since the system activation functionality is common for both systems A and B, we could implement it as part of Sys_Base. That would require changing the declaration from a simple interface to an abstract record:

```
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
with null record;
```

Now, we can add the Boolean component to the record (as a private component) and override the subprograms of the Activation_IF interface. This is the adapted declaration:

```
type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with private;

overriding procedure Activate (E : in out Sys_Base);
overriding function Is_Active (E : Sys_Base) return Boolean;
overriding procedure Deactivate (E : in out Sys_Base);

private

type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with record
    Active : Boolean;
end record;
```

72.3.3 Derived types

In the declaration of the Sys_Base type we've just seen, we're not overriding the Value function — from the Value_Retrieval_IF interface — for the Sys_Base type, so it remains an abstract function for Sys_Base. Therefore, the Sys_Base type itself remains abstract and needs to be explicitly declared as such.

We use this strategy to ensure that all types derived from Sys_Base need to implement their own version of the Value function. For example:

```
type A is new Sys_Base with private;

overriding function Value (E : A) return Float;
```

Here, the A type is derived from the Sys_Base and it includes its own version of the Value function by overriding it. Therefore, A is not an abstract type anymore and can be used to declare objects:

```
procedure Main is
    Obj : A;
    V   : Float;
begin
    Obj.Activate;
    V := Obj.Value;
end Main;
```

Important

Note that the use of the **overriding** keyword in the subprogram declaration is not strictly necessary. In fact, we could leave this keyword out, and the code would still compile. However, if provided, the compiler will check whether the information is correct.

Using the **overriding** keyword can help to avoid bad surprises — when you *may think* that you're overriding a subprogram, but you're actually not. Similarly, you can also write **not overriding** to be explicit about subprograms that are new primitives of a derived type. For example:

```
not overriding function Check (E : AB) return Boolean;
```

We also need to declare the values that are used internally in systems A and B. For system A, this is the declaration:

```

type A is new Sys_Base with private;

overriding function Value (E : A) return Float;

private

type Val_Array is array (Positive range <>) of Float;

type A is new Sys_Base with record
    Val : Val_Array (1 .. 2);
end record;

```

72.3.4 Subprograms from parent

In the previous implementation, we've seen that the A_Activate and B_Activate procedures perform the following steps:

- initialize internal values;
- indicate that the system is active (by setting the Active flag to **True**).

In the implementation of the Activate procedure for the Sys_Base type, however, we're only dealing with the second step. Therefore, we need to override the Activate procedure and make sure that we initialize internal values as well. First, we need to declare this procedure for type A:

```

type A is new Sys_Base with private;

overriding procedure Activate (E : in out A);

```

In the implementation of Activate, we should call the Activate procedure from the parent (Sys_Base) to ensure that whatever was performed for the parent will be performed in the derived type as well. For example:

```

overriding procedure Activate (E : in out A) is
begin
    E.Val := (others => 0.0);
    Sys_Base (E).Activate;    -- Calling Activate for Sys_Base type:
                            -- this call initializes the Active flag.
end;

```

Here, by writing **Sys_Base (E)**, we're performing a view conversion. Basically, we're telling the compiler to view E not as an object of type A, but of type Sys_Base. When we do this, any operation performed on this object will be done as if it was an object of Sys_Base type, which includes calling the Activate procedure of the Sys_Base type.

Important

If we write **T (Obj).Proc**, we're telling the compiler to call the Proc procedure of type T and apply it on Obj.

If we write **T'Class (Obj).Proc**, however, we're telling the compiler to dispatch the call. For example, if Obj is of derived type T2 and there's an overridden Proc procedure for type T2, then this procedure will be called instead of the Proc procedure for type T.

72.3.5 Type AB

While the implementation of systems A and B is almost straightforward, it gets more interesting in the case of system AB. Here, we have a similar API, but we don't need the activation mechanism implemented in the abstract type `Sys_Base`. Therefore, deriving from `Sys_Base` is not the best option. Instead, when declaring the AB type, we can simply use the same interfaces as we did for `Sys_Base`, but keep it independent from `Sys_Base`. For example:

```
type AB is new Activation_IF and Value_Retrieval_IF with private;

private

type AB is new Activation_IF and Value_Retrieval_IF with record
  SA : A;
  SB : B;
end record;
```

Naturally, we still need to override all the subprograms that are part of the `Activation_IF` and `Value_Retrieval_IF` interfaces. Also, we need to implement the additional `Check` function that was originally only available on system AB. Therefore, we declare these subprograms:

```
overriding procedure Activate (E : in out AB);
overriding function Is_Active (E : AB) return Boolean;
overriding procedure Deactivate (E : in out AB);

overriding function Value (E : AB) return Float;

not overriding function Check (E : AB) return Boolean;
```

72.3.6 Updated source-code

Finally, this is the complete source-code example:

[Ada]

Listing 23: simple.ads

```
1 package Simple is
2
3   type Activation_IF is interface;
4
5   procedure Activate (E : in out Activation_IF) is abstract;
6   function Is_Active (E : Activation_IF) return Boolean is abstract;
7   procedure Deactivate (E : in out Activation_IF) is abstract;
8
9   type Value_Retrieval_IF is interface;
10
11  function Value (E : Value_Retrieval_IF) return Float is abstract;
12
13  type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
14    with private;
15
16  overriding procedure Activate (E : in out Sys_Base);
17  overriding function Is_Active (E : Sys_Base) return Boolean;
18  overriding procedure Deactivate (E : in out Sys_Base);
19
20 private
```

(continues on next page)

(continued from previous page)

```

22 type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF
23   with record
24     Active : Boolean;
25   end record;
26
27 end Simple;

```

Listing 24: simple.adb

```

1 package body Simple is
2
3   overriding procedure Activate (E : in out Sys_Base) is
4 begin
5   E.Active := True;
6 end Activate;
7
8   overriding function Is_Active (E : Sys_Base) return Boolean is
9     (E.Active);
10
11  overriding procedure Deactivate (E : in out Sys_Base) is
12 begin
13   E.Active := False;
14 end Deactivate;
15
16 end Simple;

```

Listing 25: simple-system_a.ads

```

1 package Simple.System_A is
2
3   type A is new Sys_Base with private;
4
5   overriding procedure Activate (E : in out A);
6
7   overriding function Value (E : A) return Float;
8
9   private
10
11   type Val_Array is array (Positive range <>) of Float;
12
13   type A is new Sys_Base with record
14     Val : Val_Array (1 .. 2);
15   end record;
16
17 end Simple.System_A;

```

Listing 26: simple-system_a.adb

```

1 package body Simple.System_A is
2
3   procedure Activate (E : in out A) is
4 begin
5   E.Val := (others => 0.0);
6   Sys_Base (E).Activate;
7 end Activate;
8
9   function Value (E : A) return Float is
10    pragma Assert (E.Val'Length = 2);
11 begin
12   return (E.Val (1) + E.Val (2)) / 2.0;

```

(continues on next page)

(continued from previous page)

```

13  end Value;
14
15 end Simple.System_A;
```

Listing 27: simple-system_b.ads

```

1 package Simple.System_B is
2
3   type B is new Sys_Base with private;
4
5   overriding procedure Activate (E : in out B);
6
7   overriding function Value (E : B) return Float;
8
9 private
10
11  type B is new Sys_Base with record
12    Val : Float;
13  end record;
14
15 end Simple.System_B;
```

Listing 28: simple-system_b.adb

```

1 package body Simple.System_B is
2
3   procedure Activate (E : in out B) is
4 begin
5     E.Val := 0.0;
6     Sys_Base (E).Activate;
7   end Activate;
8
9   function Value (E : B) return Float is
10    (E.Val);
11
12 end Simple.System_B;
```

Listing 29: simple-system_ab.ads

```

1 with Simple.System_A; use Simple.System_A;
2 with Simple.System_B; use Simple.System_B;
3
4 package Simple.System_AB is
5
6   type AB is new Activation_IF and Value_Retrieval_IF with private;
7
8   overriding procedure Activate (E : in out AB);
9   overriding function Is_Active (E : AB) return Boolean;
10  overriding procedure Deactivate (E : in out AB);
11
12  overriding function Value (E : AB) return Float;
13
14  not overriding function Check (E : AB) return Boolean;
15
16 private
17
18  type AB is new Activation_IF and Value_Retrieval_IF with record
19    SA : A;
20    SB : B;
21  end record;
```

(continues on next page)

(continued from previous page)

```

22
23 end Simple.System_AB;

```

Listing 30: simple-system_ab.adb

```

1 package body Simple.System_AB is
2
3   procedure Activate (E : in out AB) is
4   begin
5     E.SA.Activate;
6     E.SB.Activate;
7   end Activate;
8
9   function Is_Active (E : AB) return Boolean is
10    (E.SA.Is_Active and E.SB.Is_Active);
11
12  procedure Deactivate (E : in out AB) is
13  begin
14    E.SA.Deactivate;
15    E.SB.Deactivate;
16  end Deactivate;
17
18  function Value (E : AB) return Float is
19    ((E.SA.Value + E.SB.Value) / 2.0);
20
21  function Check (E : AB) return Boolean is
22    Threshold : constant := 0.1;
23  begin
24    return abs (E.SA.Value - E.SB.Value) < Threshold;
25  end Check;
26
27 end Simple.System_AB;

```

Listing 31: main.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2
3 with Simple.System_AB; use Simple.System_AB;
4
5 procedure Main is
6
7   procedure Display_Active (E : AB) is
8   begin
9     if Is_Active (E) then
10       Put_Line ("System AB is active");
11     else
12       Put_Line ("System AB is not active");
13     end if;
14   end Display_Active;
15
16   procedure Display_Check (E : AB) is
17   begin
18     if Check (E) then
19       Put_Line ("System AB check: PASSED");
20     else
21       Put_Line ("System AB check: FAILED");
22     end if;
23   end Display_Check;
24
25   S : AB;

```

(continues on next page)

(continued from previous page)

```

26 begin
27   Put_Line ("Activating system AB... ");
28   Activate (S);
29
30   Display_Active (S);
31   Display_Check (S);
32
33   Put_Line ("Deactivating system AB... ");
34   Deactivate (S);
35
36   Display_Active (S);
37 end Main;

```

Runtime output

```

Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active

```

72.4 Further Improvements

When analyzing the complete source-code, we see that there are at least two areas that we could still improve.

72.4.1 Dispatching calls

The first issue concerns the implementation of the `Activate` procedure for types derived from `Sys_Base`. For those derived types, we're expecting that the `Activate` procedure of the parent must be called in the implementation of the overriding `Activate` procedure. For example:

```

package body Simple.System_A is

  procedure Activate (E : in out A) is
  begin
    E.Val := (others => 0.0);
    Activate (Sys_Base (E));
  end;

```

If a developer forgets to call that specific `Activate` procedure, however, the system won't work as expected. A better strategy could be the following:

- Declare a new `Activation_Reset` procedure for `Sys_Base` type.
- Make a dispatching call to the `Activation_Reset` procedure in the body of the `Activate` procedure (of the `Sys_Base` type).
- Let the derived types implement their own version of the `Activation_Reset` procedure.

This is a simplified view of the implementation using the points described above:

```

package Simple is

  type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with

```

(continues on next page)

(continued from previous page)

```

private;

not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;
end Simple;

package body Simple is

procedure Activate (E : in out Sys_Base) is
begin
    -- NOTE: calling "E.Activation_Reset" does NOT dispatch!
    -- We need to use the 'Class attribute here --- not using this
    -- attribute is an error that will be caught by the compiler.
    Sys_Base'Class (E).Activation_Reset;

    E.Active := True;
end Activate;

end Simple;

package Simple.System_A is

    type A is new Sys_Base with private;

private

    type Val_Array is array (Positive range <>) of Float;

    type A is new Sys_Base with record
        Val : Val_Array (1 .. 2);
    end record;

    overriding procedure Activation_Reset (E : in out A);

end Simple.System_A;

package body Simple.System_A is

    procedure Activation_Reset (E : in out A) is
    begin
        E.Val := (others => 0.0);
    end Activation_Reset;

end Simple.System_A;

```

An important detail is that, in the implementation of Activate, we use Sys_Base'Class to ensure that the call to Activation_Reset will dispatch. If we had just written E.Activation_Reset instead, then we would be calling the Activation_Reset procedure of Sys_Base itself, which is not what we actually want here. The compiler will catch the error if you don't do the conversion to the class-wide type, because it would otherwise be a statically-bound call to an abstract procedure, which is illegal at compile-time.

72.4.2 Dynamic allocation

The next area that we could improve is in the declaration of the system AB. In the previous implementation, we were explicitly describing the two components of that system, namely a component of type A and a component of type B:

```
type AB is new Activation_IF and Value_Retrieval_IF with record
  SA : A;
  SB : B;
end record;
```

Of course, this declaration matches the system requirements that we presented in the beginning. However, we could use strategies that make it easier to incorporate requirement changes later on. For example, we could hide this information about systems A and B by simply declaring an array of components of type `access` `Sys_Base'Class` and allocate them dynamically in the body of the package. Naturally, this approach might not be suitable for certain platforms. However, the advantage would be that, if we wanted to replace the component of type B by a new component of type C, for example, we wouldn't need to change the interface. This is how the updated declaration could look like:

```
type Sys_Base_Class_Access is access Sys_Base'Class;
type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;

type AB is limited new Activation_IF and Value_Retrieval_IF with record
  S_Array : Sys_Base_Array (1 .. 2);
end record;
```

Important

Note that we're now using the `limited` keyword in the declaration of type AB. That is necessary because we want to prevent objects of type AB being copied by assignment, which would lead to two objects having the same (dynamically allocated) subsystems A and B internally. This change requires that both `Activation_IF` and `Value_Retrieval_IF` are declared limited as well.

The body of `Activate` could then allocate those components:

```
procedure Activate (E : in out AB) is
begin
  E.S_Array := (new A, new B);
  for S of E.S_Array loop
    S.Activate;
  end loop;
end Activate;
```

And the body of `Deactivate` could deallocate them:

```
procedure Deactivate (E : in out AB) is
  procedure Free is
    new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
begin
  for S of E.S_Array loop
    S.Deactivate;
    Free (S);
  end loop;
end Deactivate;
```

72.4.3 Limited controlled types

Another approach that we could use to implement the dynamic allocation of systems A and B is to declare AB as a limited controlled type — based on the `Limited_Controlled` type of the `Ada.Finalization` package.

The `Limited_Controlled` type includes the following operations:

- `Initialize`, which is called when objects of a type derived from the `Limited_Controlled` type are being created — by declaring an object of the derived type, for example —, and
- `Finalize`, which is called when objects are being destroyed — for example, when an object gets out of scope at the end of a subprogram where it was created.

In this case, we must override those procedures, so we can use them for dynamic memory allocation. This is a simplified view of the update implementation:

```
package Simple.System_AB is

    type AB is limited new Ada.Finalization.Limited_Controlled and
        Activation_IF and Value_Retrieval_IF with private;

    overriding procedure Initialize (E : in out AB);
    overriding procedure Finalize   (E : in out AB);

end Simple.System_AB;

package body Simple.System_AB is

    overriding procedure Initialize (E : in out AB) is
    begin
        E.S_Array := (new A, new B);
    end Initialize;

    overriding procedure Finalize   (E : in out AB) is
        procedure Free is
            new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
    begin
        for S of E.S_Array loop
            Free (S);
        end loop;
    end Finalize;

end Simple.System_AB;
```

72.4.4 Updated source-code

Finally, this is the complete updated source-code example:

[Ada]

Listing 32: simple.ads

```
1  package Simple is
2
3      type Activation_IF is limited interface;
4
5          procedure Activate (E : in out Activation_IF) is abstract;
6          function Is_Active (E : Activation_IF) return Boolean is abstract;
7          procedure Deactivate (E : in out Activation_IF) is abstract;
```

(continues on next page)

(continued from previous page)

```

8   type Value_Retrieval_IF is limited interface;
9
10  function Value (E : Value_Retrieval_IF) return Float is abstract;
11
12  type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
13    private;
14
15  overriding procedure Activate (E : in out Sys_Base);
16  overriding function Is_Active (E : Sys_Base) return Boolean;
17  overriding procedure Deactivate (E : in out Sys_Base);
18
19  not overriding procedure Activation_Reset (E : in out Sys_Base) is abstract;
20
21 private
22
23  type Sys_Base is abstract new Activation_IF and Value_Retrieval_IF with
24    record
25      Active : Boolean;
26    end record;
27
28 end Simple;

```

Listing 33: simple.adb

```

1 package body Simple is
2
3   procedure Activate (E : in out Sys_Base) is
4   begin
5     -- NOTE: calling "E.Activation_Reset" does NOT dispatch!
6     --       We need to use the 'Class attribute:
7     Sys_Base'Class (E).Activation_Reset;
8
9     E.Active := True;
10    end Activate;
11
12  function Is_Active (E : Sys_Base) return Boolean is
13    (E.Active);
14
15  procedure Deactivate (E : in out Sys_Base) is
16  begin
17    E.Active := False;
18  end Deactivate;
19
20 end Simple;

```

Listing 34: simple-system_a.ads

```

1 package Simple.System_A is
2
3   type A is new Sys_Base with private;
4
5   overriding function Value (E : A) return Float;
6
7 private
8
9   type Val_Array is array (Positive range <>) of Float;
10
11  type A is new Sys_Base with record
12    Val : Val_Array (1 .. 2);

```

(continues on next page)

(continued from previous page)

```

13  end record;
14
15  overriding procedure Activation_Reset (E : in out A);
16
17 end Simple.System_A;

```

Listing 35: simple-system_a.adb

```

1 package body Simple.System_A is
2
3   procedure Activation_Reset (E : in out A) is
4   begin
5     E.Val := (others => 0.0);
6   end Activation_Reset;
7
8   function Value (E : A) return Float is
9     pragma Assert (E.Val'Length = 2);
10  begin
11    return (E.Val (1) + E.Val (2)) / 2.0;
12  end Value;
13
14 end Simple.System_A;

```

Listing 36: simple-system_b.ads

```

1 package Simple.System_B is
2
3   type B is new Sys_Base with private;
4
5   overriding function Value (E : B) return Float;
6
7   private
8
9   type B is new Sys_Base with record
10    Val : Float;
11  end record;
12
13  overriding procedure Activation_Reset (E : in out B);
14
15 end Simple.System_B;

```

Listing 37: simple-system_b.adb

```

1 package body Simple.System_B is
2
3   procedure Activation_Reset (E : in out B) is
4   begin
5     E.Val := 0.0;
6   end Activation_Reset;
7
8   function Value (E : B) return Float is
9     (E.Val);
10
11 end Simple.System_B;

```

Listing 38: simple-system_ab.ads

```

1 with Ada.Finalization;
2

```

(continues on next page)

(continued from previous page)

```

3 package Simple.System_AB is
4
5     type AB is limited new Ada.Finalization.Limited_Controlled and
6         Activation_IF and Value_Retrieval_IF with private;
7
8     overriding procedure Activate (E : in out AB);
9     overriding function Is_Active (E : AB) return Boolean;
10    overriding procedure Deactivate (E : in out AB);
11
12    overriding function Value (E : AB) return Float;
13
14    not overriding function Check (E : AB) return Boolean;
15
16 private
17
18     type Sys_Base_Class_Access is access Sys_Base'Class;
19     type Sys_Base_Array is array (Positive range <>) of Sys_Base_Class_Access;
20
21     type AB is limited new Ada.Finalization.Limited_Controlled and
22         Activation_IF and Value_Retrieval_IF with record
23         S_Array : Sys_Base_Array (1 .. 2);
24     end record;
25
26     overriding procedure Initialize (E : in out AB);
27     overriding procedure Finalize (E : in out AB);
28
29 end Simple.System_AB;

```

Listing 39: simple-system_ab.adb

```

1 with Ada.Unchecked_Deallocation;
2
3 with Simple.System_A; use Simple.System_A;
4 with Simple.System_B; use Simple.System_B;
5
6 package body Simple.System_AB is
7
8     overriding procedure Initialize (E : in out AB) is
9     begin
10        E.S_Array := (new A, new B);
11    end Initialize;
12
13    overriding procedure Finalize (E : in out AB) is
14        procedure Free is
15            new Ada.Unchecked_Deallocation (Sys_Base'Class, Sys_Base_Class_Access);
16    begin
17        for S of E.S_Array loop
18            Free (S);
19        end loop;
20    end Finalize;
21
22    procedure Activate (E : in out AB) is
23    begin
24        for S of E.S_Array loop
25            S.Activate;
26        end loop;
27    end Activate;
28
29    function Is_Active (E : AB) return Boolean is
30        (for all S of E.S_Array => S.Is_Active);
31

```

(continues on next page)

(continued from previous page)

```

32  procedure Deactivate (E : in out AB) is
33  begin
34      for S of E.S_Array loop
35          S.Deactivate;
36      end loop;
37  end Deactivate;

38
39  function Value (E : AB) return Float is
40      ((E.S_Array (1).Value + E.S_Array (2).Value) / 2.0);

41
42  function Check (E : AB) return Boolean is
43      Threshold : constant := 0.1;
44  begin
45      return abs (E.S_Array (1).Value - E.S_Array (2).Value) < Threshold;
46  end Check;

47
48 end Simple.System_AB;

```

Listing 40: main.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Simple.System_AB; use Simple.System_AB;
4
5  procedure Main is
6
7      procedure Display_Active (E : AB) is
8  begin
9      if Is_Active (E) then
10         Put_Line ("System AB is active");
11     else
12         Put_Line ("System AB is not active");
13     end if;
14  end Display_Active;
15
16  procedure Display_Check (E : AB) is
17  begin
18      if Check (E) then
19         Put_Line ("System AB check: PASSED");
20     else
21         Put_Line ("System AB check: FAILED");
22     end if;
23  end Display_Check;
24
25  S : AB;
26  begin
27      Put_Line ("Activating system AB...");
28      Activate (S);
29
30      Display_Active (S);
31      Display_Check (S);
32
33      Put_Line ("Deactivating system AB...");
34      Deactivate (S);
35
36      Display_Active (S);
37  end Main;

```

Runtime output

```
Activating system AB...
System AB is active
System AB check: PASSED
Deactivating system AB...
System AB is not active
```

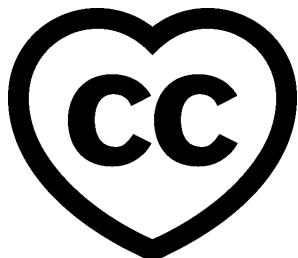
Naturally, this is by no means the best possible implementation of system AB. By applying other software design strategies that we haven't covered here, we could most probably think of different ways to use object-oriented programming to improve this implementation. Also, in comparison to the [original implementation](#) (page 841), we recognize that the amount of source-code has grown. On the other hand, we now have a system that is factored nicely, and also more extensible.

Part VII

SPARK Ada for the MISRA C Developer

Copyright © 2018 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page¹¹⁰](#)



This book presents the SPARK technology — the SPARK subset of Ada and its supporting static analysis tools — through an example-driven comparison with the rules in the widely known MISRA C subset of the C language.

This document was prepared by Yannick Moy, with contributions and review from Ben Bros-gol.

¹¹⁰ <http://creativecommons.org/licenses/by-sa/4.0>

CHAPTER SEVENTYTHREE

PREFACE

MISRA C appeared in 1998 as a coding standard for C; it focused on avoiding error-prone programming features of the C programming language rather than on enforcing a particular programming style. A study of coding standards for C by [Les Hatton¹¹¹](#) found that, compared to ten typical coding standards for C, MISRA C was the only one to focus exclusively on error avoidance rather than style enforcement, and by a very large margin.

The popularity of the C programming language, as well as its many traps and pitfalls, have led to the huge success of MISRA C in domains where C is used for high-integrity software. This success has driven tool vendors to propose many competing implementations of [MISRA C¹¹²](#) checkers. Tools compete in particular on the coverage of MISRA C guidelines that they help enforce, as it is impossible to enforce the 16 directives and 143 rules (collectively referred to as guidelines) of MISRA C.

The 16 directives are broad guidelines, and it is not possible to define compliance in a unique and automated way. For example, "*all code should be traceable to documented requirements*" (Directive 3.1). Thus no tool is expected to enforce directives, as the MISRA C:2012 states in introduction to the guidelines: "*different tools may place widely different interpretations on what constitutes a non-compliance.*"

The 143 rules on the contrary are completely and precisely defined, and "*static analysis tools should be capable of checking compliance with rules*". But the same sentence continues with "*subject to the limitations described in Section 6.5*", which addresses "decidability of rules". It turns out that 27 rules out of 143 are not decidable, so no tool can always detect all violations of these rules without at the same time reporting "false alarms" on code that does not constitute a violation.

An example of an undecidable rule is rule 1.3: "*There shall be no occurrence of undefined or critical unspecified behaviour.*" Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable. For the most part, MISRA C checkers ignore undecidable rules such as rule 1.3 and instead focus on the 116 rules for which detection of violations can be automated. It is telling in that respect that the MISRA C:2012 document and its accompanying set of examples (which can be downloaded from the [MISRA website¹¹³](#)) does not provide any example for rule 1.3.

However, violations of undecidable rules such as rule 1.3 are known to have dramatic impact on software quality. Violations of rule 1.3 in particular are commonly amplified by compilers using the permission in the C standard to optimize aggressively without looking at the consequences for programs with undefined or critical unspecified behavior. It would be valid to ignore these rules if violations did not occur in practice, but on the contrary even experienced programmers write C code with undefined or critical unspecified behavior. An example comes from the MISRA C Committee itself in its "Appendix I: Example deviation record" of the MISRA C:2012 document, repeated in "Appendix A: Example deviation record" of the [MISRA C: Compliance 2016 document¹¹⁴](#), where the following code is proposed as

¹¹¹ <https://www.leshatton.org/Documents/MISRAC.pdf>

¹¹² https://en.wikipedia.org/wiki/MISRA_C

¹¹³ <https://www.misra.org.uk>

¹¹⁴ https://www.misra.org.uk/LinkClick.aspx?fileticket=w_Syhpkf7xA%3d&tabid=57

a deviation of rule 10.6 "*The value of a composite expression shall not be assigned to an object with wider essential type*":

```
uint32_t prod = qty * time_step;
```

Here, the multiplication of two unsigned 16-bit values and assignment of the result to an unsigned 32-bit variable constitutes a violation of the aforementioned rule, which gets justified for efficiency reasons. What the authors seem to have missed is that the multiplication is then performed with the signed integer type `int` instead of the target unsigned type `uint32_t`. Thus the multiplication of two unsigned 16-bit values may lead to an overflow of the 32-bit intermediate signed result, which is an occurrence of an undefined behavior. In such a case, a compiler is free to assume that the value of `prod` cannot exceed $2^{31} - 1$ (the maximal value of a signed 32-bit integer) as otherwise an undefined behavior would have been triggered. For example, the undefined behavior with values 65535 for `qty` and `time_step` is reported when running the code compiled by either the GCC or LLVM compiler with option `-fsanitize=undefined`.

The MISRA C checkers that detect violations of undecidable rules are either unsound tools that can detect only some of the violations, or sound tools that guarantee to detect all such violations at the cost of possibly many false reports of violations. This is a direct consequence of undecidability. However, static analysis technology is available that can achieve soundness without inundating users with false alarms. One example is the SPARK toolset developed by AdaCore, Altran and Inria, which is based on four principles:

- The base language Ada provides a solid foundation for static analysis through a well-defined language standard, strong typing and rich specification features.
- The SPARK subset of Ada restricts the base language in essential ways to support static analysis, by controlling sources of ambiguity such as side-effects and aliasing.
- The static analysis tools work mostly at the granularity of an individual function, making the analysis more precise and minimizing the possibility of false alarms.
- The static analysis tools are interactive, allowing users to guide the analysis if necessary or desired.

In this document, we show how SPARK can be used to achieve high code quality with guarantees that go beyond what would be feasible with MISRA C.

An on-line and interactive version of this document is available at [AdaCore's learn.adacore.com site¹¹⁵](https://learn.adacore.com).

¹¹⁵ https://learn.adacore.com/courses/SPARK_for_the_MISRA_C_Developer

ENFORCING BASIC PROGRAM CONSISTENCY

Many consistency properties that are taken for granted in other languages are not enforced in C. The basic property that all uses of a variable or function are consistent with its type is not enforced by the language and is also very difficult to enforce by a tool. Three features of C contribute to that situation:

- the textual-based inclusion of files means that every included declaration is subject to a possibly different reinterpretation depending on context.
- the lack of consistency requirements across translation units means that type inconsistencies can only be detected at link time, something linkers are ill-equipped to do.
- the default of making a declaration externally visible means that declarations that should be local will be visible to the rest of the program, increasing the chances for inconsistencies.

MISRA C contains guidelines on all three fronts to enforce basic program consistency.

74.1 Taming Text-Based Inclusion

The text-based inclusion of files is one of the dated idiosyncrasies of the C programming language that was inherited by C++ and that is known to cause quality problems, especially during maintenance. Although multiple inclusion of a file in the same translation unit can be used to emulate template programming, it is generally undesirable. Indeed, MISRA C defines Directive 4.10 precisely to forbid it for header files: "*Precautions shall be taken in order to prevent the contents of a header file being included more than once*".

The subsequent section on "Preprocessing Directives" contains 14 rules restricting the use of text-based inclusion through preprocessing. Among other things these rules forbid the use of the `#undef` directive (which works around conflicts in macro definitions introduced by text-based inclusion) and enforces the well-known practice of enclosing macro arguments in parentheses (to avoid syntactic reinterpretations in the context of the macro use).

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context instead of textual inclusion of content, using `with` clauses:

Listing 1: hello_world.adb

```
1  with Ada.Text_IO;
2
3  procedure Hello_World is
4    begin
5      Ada.Text_IO.Put_Line ("hello, world!");
6    end Hello_World;
```

Runtime output

```
hello, world!
```

Note that **with** clauses are only allowed at the beginning of files; the compiler issues an error if they are used elsewhere:

Listing 2: hello_world.adb

```
1 procedure Hello_World is
2     with Ada.Text_IO; -- Illegal
3 begin
4     Ada.Text_IO.Put_Line ("hello, world!");
5 end Hello_World;
```

Importing a unit (i.e., specifying it in a **with** clause) multiple times is harmless, as it is equivalent to importing it once, but a compiler warning lets us know about the redundancy:

Listing 3: hello_world.adb

```
1 with Ada.Text_IO;
2 with Ada.Text_IO; -- Legal but useless
3
4 procedure Hello_World is
5 begin
6     Ada.Text_IO.Put_Line ("hello, world!");
7 end Hello_World;
```

Build output

```
hello_world.adb:2:06: warning: redundant with clause [-gnatwr]
```

Runtime output

```
hello, world!
```

The order in which units are imported is irrelevant. All orders are valid and have the same semantics.

No conflict arises from importing multiple units, even if the same name is defined in several, since each unit serves as namespace for the entities which it defines. So we can define our own version of `Put_Line` in some `Helper` unit and import it together with the standard version defined in `Ada.Text_IO`:

Listing 4: helper.ads

```
1 package Helper is
2     procedure Put_Line (S : String);
3 end Helper;
```

Listing 5: helper.adb

```
1 with Ada.Text_IO;
2
3 package body Helper is
4     procedure Put_Line (S : String) is
5 begin
6     Ada.Text_IO.Put_Line ("Start helper version");
7     Ada.Text_IO.Put_Line (S);
8     Ada.Text_IO.Put_Line ("End helper version");
9     end Put_Line;
10 end Helper;
```

Listing 6: hello_world.adb

```

1  with Ada.Text_IO;
2  with Helper;
3
4  procedure Hello_World is
5  begin
6      Ada.Text_IO.Put_Line ("hello, world!");
7      Helper.Put_Line ("hello, world!");
8  end Hello_World;

```

Runtime output

```

hello, world!
Start helper version
hello, world!
End helper version

```

The only way a conflict can arise is if we want to be able to reference Put_Line directly, without using the qualified name Ada.Text_IO.Put_Line or Helper.Put_Line. The **use** clause makes public declarations from a unit available directly:

Listing 7: helper.ads

```

1  package Helper is
2      procedure Put_Line (S : String);
3  end Helper;

```

Listing 8: helper.adb

```

1  with Ada.Text_IO;
2
3  package body Helper is
4      procedure Put_Line (S : String) is
5      begin
6          Ada.Text_IO.Put_Line ("Start helper version");
7          Ada.Text_IO.Put_Line (S);
8          Ada.Text_IO.Put_Line ("End helper version");
9      end Put_Line;
10 end Helper;

```

Listing 9: hello_world.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Helper; use Helper;
3
4  procedure Hello_World is
5  begin
6      Ada.Text_IO.Put_Line ("hello, world!");
7      Helper.Put_Line ("hello, world!");
8      Put_Line ("hello, world!"); -- ERROR
9  end Hello_World;

```

Build output

```

hello_world.adb:8:04: error: ambiguous expression (cannot resolve "Put_Line")
hello_world.adb:8:04: error: possible interpretation at helper.ads:2
hello_world.adb:8:04: error: possible interpretation at a-textio.ads:507
gprbuild: *** compilation phase failed

```

Here, both units Ada.Text_IO and Helper define a procedure Put_Line taking a **String** as

argument, so the compiler cannot disambiguate the direct call to Put_Line and issues an error.

Note that it helpfully points to candidate declarations, so that the user can decide which qualified name to use as in the previous two calls.

Issues arising in C as a result of text-based inclusion of files are thus completely prevented in SPARK (and Ada) thanks to semantic import of units. Note that the C++ committee identified this weakness some time ago and has approved¹¹⁶ the addition of *modules* to C++20, which provide a mechanism for semantic import of units.

74.2 Hardening Link-Time Checking

An issue related to text-based inclusion of files is that there is no single source for declaring the type of a variable or function. If a file origin.c defines a variable var and functions fun and print:

Listing 10: origin.c

```
1 #include <stdio.h>
2
3 int var = 0;
4 int fun() {
5     return 1;
6 }
7 void print() {
8     printf("var = %d\n", var);
9 }
```

and the corresponding header file origin.h declares var, fun and print as having external linkage:

Listing 11: origin.h

```
1 extern int var;
2 extern int fun();
3 extern void print();
```

then client code can include origin.h with declarations for var and fun:

Listing 12: main.c

```
1 #include "origin.h"
2
3 int main() {
4     var = fun();
5     print();
6     return 0;
7 }
```

Runtime output

```
var = 1
```

or, equivalently, repeat these declarations directly:

¹¹⁶ <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf>

Listing 13: main.c

```

1 extern int var;
2 extern int fun();
3 extern void print();

4
5 int main() {
6     var = fun();
7     print();
8     return 0;
9 }
```

Runtime output

var = 1

Then, if an inconsistency is introduced in the type of var or fun between these alternative declarations and their actual type, the compiler cannot detect it. Only the linker, which has access to the set of object files for a program, can detect such inconsistencies. However, a linker's main task is to link, not to detect inconsistencies, and so inconsistencies in the type of variables and functions in most cases cannot be detected. For example, most linkers cannot detect if the type of var or the return type of fun is changed to **float** in the declarations above. With the declaration of var changed to **float**, the above program compiles and runs without errors, producing the erroneous output var = 1065353216 instead of var = 1. With the return type of fun changed to **float** instead, the program still compiles and runs without errors, producing this time the erroneous output var = 0.

The inconsistency just discussed is prevented by MISRA C Rule 8.3 "*All declarations of an object or function shall use the same names and type qualifiers*". This is a decidable rule, but it must be enforced at system level, looking at all translation units of the complete program. MISRA C Rule 8.6 also requires a unique definition for a given identifier across translation units, and Rule 8.5 requires that an external declaration shared between translation units comes from the same file. There is even a specific section on "Identifiers" containing 9 rules requiring uniqueness of various categories of identifiers.

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context using **with** clauses to provide a unique declaration for each entity.

74.3 Going Towards Encapsulation

Many problems in C stem from the lack of encapsulation. There is no notion of namespace that would allow a file to make its declarations available without risking a conflict with other files. Thus MISRA C has a number of guidelines that discourage the use of external declarations:

- Directive 4.8 encourages hiding the definition of structures and unions in implementation files (.c files) when possible: "*If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.*"
- Rule 8.7 forbids the use of external declarations when not needed: "*Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.*"
- Rule 8.8 forces the explicit use of keyword **static** when appropriate: "*The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.*"

The basic unit of modularization in SPARK, as in Ada, is the *package*. A package always has a spec (in an .ads file), which defines the interface to other units. It generally also has a body (in an .adb file), which completes the spec with an implementation. Only declarations from the package spec are visible from other units when they import (**with**) the package. In fact, only declarations from what is called the "visible part" of the spec (before the keyword **private**) are visible from units that **with** the package.

Listing 14: helper.ads

```
1 package Helper is
2     procedure Public_Put_Line (S : String);
3 private
4     procedure Private_Put_Line (S : String);
5 end Helper;
```

Listing 15: helper.adb

```
1 with Ada.Text_IO;
2
3 package body Helper is
4     procedure Public_Put_Line (S : String) is
5 begin
6     Ada.Text_IO.Put_Line (S);
7 end Public_Put_Line;
8
9     procedure Private_Put_Line (S : String) is
10 begin
11     Ada.Text_IO.Put_Line (S);
12 end Private_Put_Line;
13
14     procedure Body_Put_Line (S : String) is
15 begin
16     Ada.Text_IO.Put_Line (S);
17 end Body_Put_Line;
18 end Helper;
```

Listing 16: hello_world.adb

```
1 with Helper; use Helper;
2
3 procedure Hello_World is
4 begin
5     Public_Put_Line ("hello, world!");
6     Private_Put_Line ("hello, world!"); -- ERROR
7     Body_Put_Line ("hello, world!"); -- ERROR
8 end Hello_World;
```

Build output

```
hello_world.adb:6:04: error: "Private_Put_Line" is not visible
hello_world.adb:6:04: error: non-visible (private) declaration at helper.ads:4
hello_world.adb:7:04: error: "Body_Put_Line" is undefined
gprbuild: *** compilation phase failed
```

Note the different errors on the calls to the private and body versions of Put_Line. In the first case the compiler can locate the candidate procedure but it is illegal to call it from Hello_World, in the second case the compiler does not even know about any Body_Put_Line when compiling Hello_World since it only looks at the spec and not the body.

SPARK (and Ada) also allow defining a type in the private part of a package spec while simply declaring the type name in the public ("visible") part of the spec. This way, client

code — i.e., code that **with**'s the package — can use the type, typically through a public API, but have no access to how the type is implemented:

Listing 17: vault.ads

```

1 package Vault is
2   type Data is private;
3   function Get (X : Data) return Integer;
4   procedure Set (X : out Data; Value : Integer);
5 private
6   type Data is record
7     Val : Integer;
8   end record;
9 end Vault;
```

Listing 18: vault.adb

```

1 package body Vault is
2   function Get (X : Data) return Integer is (X.Val);
3   procedure Set (X : out Data; Value : Integer) is
4 begin
5   X.Val := Value;
6 end Set;
7 end Vault;
```

Listing 19: information_system.ads

```

1 with Vault;
2
3 package Information_System is
4   Archive : Vault.Data;
5 end Information_System;
```

Listing 20: hacker.adb

```

1 with Information_System;
2 with Vault;
3
4 procedure Hacker is
5   V : Integer := Vault.Get (Information_System.Archive);
6 begin
7   Vault.Set (Information_System.Archive, V + 1);
8   Information_System.Archive.Val := 0; -- ERROR
9 end Hacker;
```

Build output

```

hacker.adb:8:22: error: invalid prefix in selected component "Information_System.
  ^Archive"
gprbuild: *** compilation phase failed
```

Note that it is possible to declare a variable of type `Vault.Data` in package `Information_System` and to get/set it through its API in procedure `Hacker`, but not to directly access its `Val` field.

ENFORCING BASIC SYNTACTIC GUARANTEES

C's syntax is concise but also very permissive, which makes it easy to write programs whose effect is not what was intended. MISRA C contains guidelines to:

- clearly distinguish code from comments
- specially handle function parameters and result
- ensure that control structures are not abused

75.1 Distinguishing Code and Comments

The problem arises from block comments in C, starting with /* and ending with */. These comments do not nest with other block comments or with line comments. For example, consider a block comment surrounding three lines that each increase variable a by one:

```
/*
++a;
++a;
++a; */
```

Now consider what happens if the first line is commented out using a block comment and the third line is commented out using a line comment (also known as a C++ style comment, allowed in C since C99):

```
/*
/* ++a; */
++a;
// ++a; */
```

The result of commenting out code that was already commented out is that the second line of code becomes live! Of course, the above example is simplified, but similar situations do arise in practice, which is the reason for MISRA C Directive 4.1 "*Sections of code should not be 'commented out'*". This is reinforced with Rules 3.1 and 3.2 from the section on "Comments" that forbid in particular the use of /* inside a comment like we did above.

These situations cannot arise in SPARK (or in Ada), as only line comments are permitted, using --:

```
-- A := A + 1;
-- A := A + 1;
-- A := A + 1;
```

So commenting again the first and third lines does not change the effect:

```
-- -- A := A + 1;
-- A := A + 1;
-- -- A := A + 1;
```

75.2 Specially Handling Function Parameters and Result

75.2.1 Handling the Result of Function Calls

It is possible in C to ignore the result of a function call, either implicitly or else explicitly by converting the result to **void**:

```
f();
(void)f();
```

This is particularly dangerous when the function returns an error status, as the caller is then ignoring the possibility of errors in the callee. Thus the MISRA C Directive 4.7: "*If a function returns error information, then that error information shall be tested*". In the general case of a function returning a result which is not an error status, MISRA C Rule 17.7 states that "*The value returned by a function having non-void return type shall be used*", where an explicit conversion to **void** counts as a use.

In SPARK, as in Ada, the result of a function call must be used, for example by assigning it to a variable or by passing it as a parameter, in contrast with procedures (which are equivalent to void-returning functions in C). SPARK analysis also checks that the result of the function is actually used to influence an output of the calling subprogram. For example, the first two calls to F in the following are detected as unused, even though the result of the function call is assigned to a variable, which is itself used in the second case:

Listing 1: fun.ads

```
1 package Fun is
2     function F return Integer is (1);
3 end Fun;
```

Listing 2: use_f.adb

```
1 with Fun; use Fun;
2
3 procedure Use_F (Z : out Integer) is
4     X, Y : Integer;
5 begin
6     X := F;
7
8     Y := F;
9     X := Y;
10
11    Z := F;
12 end Use_F;
```

Build output

```
use_f.adb:4:04: warning: variable "X" is assigned but never read [-gnatwm]
use_f.adb:6:04: warning: useless assignment to "X", value overwritten at line 9 [-gnatwm]
use_f.adb:9:04: warning: possibly useless assignment to "X", value might not be referenced [-gnatwm]
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
use_f.adb:4:04: warning: variable "X" is assigned but never read [-gnatwm]
use_f.adb:6:04: warning: useless assignment to "X", value overwritten at line 9 [-gnatwm]
```

(continues on next page)

(continued from previous page)

```
use_f.adb:6:06: warning: unused assignment
use_f.adb:8:06: warning: unused assignment
use_f.adb:9:04: warning: possibly useless assignment to "X", value might not be ↴
referenced [-gnatwm]
use_f.adb:9:06: warning: unused assignment
```

Only the result of the third call is used to influence the value of an output of Use_F, here the output parameter Z of the procedure.

75.2.2 Handling Function Parameters

In C, function parameters are treated as local variables of the function. They can be modified, but these modifications won't be visible outside the function. This is an opportunity for mistakes. For example, the following code, which appears to swap the values of its parameters, has in reality no effect:

```
void swap (int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

MISRA C Rule 17.8 prevents such mistakes by stating that "*A function parameter should not be modified*".

No such rule is needed in SPARK, since function parameters are only inputs so cannot be modified, and procedure parameters have a *mode* defining whether they can be modified or not. Only parameters of mode **out** or **ada:in out** can be modified — and these are prohibited from functions in SPARK — and their modification is visible at the calling site. For example, assigning to a parameter of mode **in** (the default parameter mode if omitted) results in compilation errors:

Listing 3: swap.ads

```
1 procedure Swap (X, Y : Integer);
```

Listing 4: swap.adb

```
1 procedure Swap (X, Y : Integer) is
2     Tmp : Integer := X;
3 begin
4     X := Y; -- ERROR
5     Y := Tmp; -- ERROR
6 end Swap;
```

Build output

```
swap.adb:4:04: error: assignment to "in" mode parameter not allowed
swap.adb:5:04: error: assignment to "in" mode parameter not allowed
gprbuild: *** compilation phase failed
```

Here is the output of AdaCore's GNAT compiler:

```
1.      procedure Swap (X, Y : Integer) is
2.          Tmp : Integer := X;
3.      begin
4.          X := Y; -- ERROR
|
```

(continues on next page)

(continued from previous page)

```
>>> assignment to "in" mode parameter not allowed  
5.      Y := Tmp;  --  ERROR  
|  
>>> assignment to "in" mode parameter not allowed  
6.  end Swap;
```

The correct version of Swap in SPARK takes parameters of mode **in out**:

Listing 5: swap.ads

```
1  procedure Swap (X, Y : in out Integer);
```

Listing 6: swap.adb

```
1  procedure Swap (X, Y : in out Integer) is  
2    Tmp : constant Integer := X;  
3  begin  
4    X := Y;  
5    Y := Tmp;  
6  end Swap;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...  
Phase 2 of 2: analysis of data and information flow ...
```

75.3 Ensuring Control Structures Are Not Abused

The previous issue (ignoring the result of a function call) is an example of a control structure being abused, due to the permissive syntax of C. There are many such examples, and MISRA C contains a number of guidelines to prevent such abuse.

75.3.1 Preventing the Semicolon Mistake

Because a semicolon can act as a statement, and because an if-statement and a loop accept a simple statement (possibly only a semicolon) as body, inserting a single semicolon can completely change the behavior of the code:

```
int func() {  
    if (0)  
        return 1;  
    while (1)  
        return 0;  
    return 0;  
}
```

As written, the code above returns with status 0. If a semicolon is added after the first line (**if (0);**), then the code returns with status 1. If a semicolon is added instead after the third line (**while (1);**), then the code does not return. To prevent such surprises, MISRA C Rule 15.6 states that "*The body of an iteration-statement or a selection-statement shall be a compound statement*" so that the code above must be written:

```

int func() {
    if (0) {
        return 1;
    }
    while (1) {
        return 0;
    }
    return 0;
}

```

Note that adding a semicolon after the test of the `if` or `while` statement has the same effect as before! But doing so would violate MISRA C Rule 15.6.

In SPARK, the semicolon is not a statement by itself, but rather a marker that terminates a statement. The null statement is an explicit `null;`, and all blocks of statements have explicit `begin` and `end` markers, which prevents mistakes that are possible in C. The SPARK (also Ada) version of the above C code is as follows:

Listing 7: func.ads

```

1 function Func return Integer;

```

Listing 8: func.adb

```

1 function Func return Integer is
2 begin
3     if False then
4         return 1;
5     end if;
6     while True loop
7         return 0;
8     end loop;
9     return 0;
10 end Func;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
func.adb:3:04: warning: statement has no effect
func.adb:4:07: warning: this statement is never reached

```

75.3.2 Avoiding Complex Switch Statements

Switch statements are well-known for being easily misused. Control can jump to any case section in the body of the switch, which in C can be before any statement contained in the body of the switch. At the end of the sequence of statements associated with a case, execution continues with the code that follows unless a break is encountered. This is a recipe for mistakes, and MISRA C enforces a simpler *well-formed* syntax for switch statements defined in Rule 16.1: "All switch statements shall be well-formed".

The other rules in the section on "Switch statements" go on detailing individual consequences of Rule 16.1. For example Rule 16.3 forbids the fall-through from one case to the next: "*An unconditional break statement shall terminate every switch-clause*". As another example, Rule 16.4 mandates the presence of a default case to handle cases not taken into account explicitly: "*Every switch statement shall have a default label*".

The analog of the C switch statements in SPARK (and in Ada) is the case statement. This statement has a simpler and more robust structure than the C switch, with control automatically exiting after one of the case alternatives is executed, and the compiler checking

that the alternatives are disjoint (like in C) and complete (unlike in C). So the following code is rejected by the compiler:

Listing 9: sign_domain.ads

```

1 package Sign_Domain is
2
3     type Sign is (Negative, Zero, Positive);
4
5     function Opposite (A : Sign) return Sign is
6         (case A is -- ERROR
7             when Negative => Positive,
8             when Positive => Negative);
9
10    function Multiply (A, B : Sign) return Sign is
11        (case A is
12            when Negative      => Opposite (B),
13            when Zero | Positive => Zero,
14            when Positive      => B); -- ERROR
15
16    procedure Get_Sign (X : Integer; S : out Sign);
17
18 end Sign_Domain;

```

Listing 10: sign_domain.adb

```

1 package body Sign_Domain is
2
3     procedure Get_Sign (X : Integer; S : out Sign) is
4         begin
5             case X is
6                 when 0 => S := Zero;
7                 when others => S := Negative; -- ERROR
8                 when 1 .. Integer'Last => S := Positive;
9             end case;
10        end Get_Sign;
11
12 end Sign_Domain;

```

Build output

```

sign_domain.adb:7:15: error: the choice "others" must appear alone and last
sign_domain.ads:6:07: error: missing case value: "Zero"
sign_domain.ads:14:15: error: duplication of choice value: "Positive" at line 13
gprbuild: *** compilation phase failed

```

The error in function Opposite is that the **when** choices do not cover all values of the target expression. Here, A is of the enumeration type Sign, so all three values of the enumeration must be covered.

The error in function Multiply is that **Positive** is covered twice, in the second and the third alternatives. This is not allowed.

The error in procedure Get_Sign is that the **others** choice (the equivalent of C **default** case) must come last. Note that an **others** choice would be useless in Opposite and Multiply, as all Sign values are covered.

Here is a correct version of the same code:

Listing 11: sign_domain.ads

```

1 package Sign_Domain is
2

```

(continues on next page)

(continued from previous page)

```

3  type Sign is (Negative, Zero, Positive);
4
5  function Opposite (A : Sign) return Sign is
6    (case A is
7      when Negative => Positive,
8      when Zero      => Zero,
9      when Positive => Negative);
10
11 function Multiply (A, B : Sign) return Sign is
12   (case A is
13     when Negative => Opposite (B),
14     when Zero      => Zero,
15     when Positive => B);
16
17 procedure Get_Sign (X : Integer; S : out Sign);
18
19 end Sign_Domain;

```

Listing 12: sign_domain.adb

```

1 package body Sign_Domain is
2
3   procedure Get_Sign (X : Integer; S : out Sign) is
4   begin
5     case X is
6       when 0 => S := Zero;
7       when 1 .. Integer'Last => S := Positive;
8       when others => S := Negative;
9     end case;
10    end Get_Sign;
11
12 end Sign_Domain;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
sign_domain.ads:17:37: info: initialization of "S" proved

```

75.3.3 Avoiding Complex Loops

Similarly to C switches, for-loops in C can become unreadable. MISRA C thus enforces a simpler *well-formed* syntax for for-loops, defined in Rule 14.2: "*A for loop shall be well-formed*". The main effect of this simplification is that for-loops in C look like for-loops in SPARK (and in Ada), with a *loop counter* that is incremented or decremented at each iteration. Section 8.14 defines precisely what a loop counter is:

1. It has a scalar type;
2. Its value varies monotonically on each loop iteration; and
3. It is used in a decision to exit the loop.

In particular, Rule 14.2 forbids any modification of the loop counter inside the loop body. Here's the example used in MISRA C:2012 to illustrate this rule:

```

bool_t flag = false;

for ( int16_t i = 0; ( i < 5 ) && !flag; i++ )

```

(continues on next page)

(continued from previous page)

```
{
  if ( C )
  {
    flag = true; /* Compliant - allows early termination of loop */
  }

  i = i + 3;      /* Non-compliant - altering the loop counter */
}
```

The equivalent SPARK (and Ada) code does not compile, because of the attempt to modify the value of the loop counter:

Listing 13: well_formed_loop.adb

```

1  procedure Well_Formed_Loop (C : Boolean) is
2    Flag : Boolean := False;
3  begin
4    for I in 0 .. 4 loop
5      exit when not Flag;
6
7      if C then
8        Flag := True;
9      end if;
10
11     I := I + 3;  -- ERROR
12   end loop;
13 end Well_Formed_Loop;
```

Build output

```
well_formed_loop.adb:11:07: error: assignment to loop parameter not allowed
gprbuild: *** compilation phase failed
```

Removing the problematic line leads to a valid program. Note that the additional condition being tested in the C for-loop has been moved to a separate exit statement at the start of the loop body.

SPARK (and Ada) loops can increase (or, with explicit syntax, decrease) the loop counter by 1 at each iteration.

```
for I in reverse 0 .. 4 loop
  ...
  -- Successive values of I are 4, 3, 2, 1, 0
end loop;
```

SPARK loops can iterate over any discrete type; i.e., integers as above or enumerations:

```
type Sign is (Negative, Zero, Positive);

for S in Sign loop
  ...
end loop;
```

75.3.4 Avoiding the Dangling Else Issue

C does not provide a closing symbol for an if-statement. This makes it possible to write the following code, which appears to try to return the absolute value of its argument, while it actually does the opposite:

Listing 14: main.c

```

1 #include <stdio.h>
2
3 int absval (int x) {
4     int result = x;
5     if (x >= 0)
6         if (x == 0)
7             result = 0;
8     else
9         result = -x;
10    return result;
11 }
12
13 int main() {
14     printf("absval(5) = %d\n", absval(5));
15     printf("absval(0) = %d\n", absval(0));
16     printf("absval(-10) = %d\n", absval(-10));
17 }
```

Runtime output

```
absval(5) = -5
absval(0) = 0
absval(-10) = -10
```

The warning issued by GCC or LLVM with option `-Wdangling-else` (implied by `-Wall`) gives a clue about the problem: although the `else` branch is written as though it completes the outer if-statement, in fact it completes the inner if-statement.

MISRA C Rule 15.6 avoids the problem: "*The body of an iteration-statement or a selection-statement shall be a compound statement*". That's the same rule as the one shown earlier for [Preventing the Semicolon Mistake](#) (page 882). So the code for `absval` must be written:

Listing 15: main.c

```

1 #include <stdio.h>
2
3 int absval (int x) {
4     int result = x;
5     if (x >= 0) {
6         if (x == 0)
7             result = 0;
8     }
9     } else {
10         result = -x;
11     }
12    return result;
13 }
14
15 int main() {
16     printf("absval(5) = %d\n", absval(5));
17     printf("absval(0) = %d\n", absval(0));
18     printf("absval(-10) = %d\n", absval(-10));
19 }
```

Runtime output

```
absval(5) = 5
absval(0) = 0
absval(-10) = 10
```

which has the expected behavior.

In SPARK (as in Ada), each if-statement has a matching end marker end **if**; so the dangling-else problem cannot arise. The above C code is written as follows:

Listing 16: absval.ads

```
1 function Absval (X : Integer) return Integer;
```

Listing 17: absval.adb

```
1 function Absval (X : Integer) return Integer is
2     Result : Integer := X;
3 begin
4     if X >= 0 then
5         if X = 0 then
6             Result := 0;
7             end if;
8         else
9             Result := -X;
10            end if;
11        return Result;
12    end Absval;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
absval.adb:9:17: medium: overflow check might fail, cannot prove upper bound for -
  ↵X [reason for check: result of negation must fit in a 32-bits machine integer] ↵
  ↵[possible fix: add precondition (-X in Integer) to subprogram at absval.ads:1]
gnatprove: unproved check messages considered as errors
```

Interestingly, SPARK analysis detects here that the negation operation on line 9 might overflow. That's an example of runtime error detection which will be covered in the chapter on *Detecting Undefined Behavior* (page 919).

ENFORCING STRONG TYPING

Annex C of MISRA C:2012 summarizes the problem succinctly:

"ISO C may be considered to exhibit poor type safety as it permits a wide range of implicit type conversions to take place. These type conversions can compromise safety as their implementation-defined aspects can cause developer confusion."

The most severe consequences come from inappropriate conversions involving pointer types, as they can cause memory safety violations. Two sections of MISRA C are dedicated to these issues: "Pointer type conversions" (9 rules) and "Pointers and arrays" (8 rules).

Inappropriate conversions between scalar types are only slightly less severe, as they may introduce arbitrary violations of the intended functionality. MISRA C has gone to great lengths to improve the situation, by defining a stricter type system on top of the C language. This is described in Appendix D of MISRA C:2012 and in the dedicated section on "The essential type model" (8 rules).

76.1 Enforcing Strong Typing for Pointers

Pointers in C provide a low-level view of the addressable memory as a set of integer addresses. To write at address 42, just go through a pointer:

Listing 1: main.c

```
1 int main() {
2     int *p = 42;
3     *p = 0;
4     return 0;
5 }
```

Running this program is likely to hit a segmentation fault on an operating system, or to cause havoc in an embedded system, both because address 42 will not be correctly aligned on a 32-bit or 64-bit machine and because this address is unlikely to correspond to valid addressable data for the application. The compiler might issue a helpful warning on the above code (with option `-Wint-conversion` implied by `-Wall` in GCC or LLVM), but note that the warning disappears when explicitly converting value 42 to the target pointer type, although the problem is still present.

Beyond their ability to denote memory addresses, pointers are also used in C to pass references as inputs or outputs to function calls, to construct complex data structures with indirection or sharing, and to denote arrays of elements. Pointers are thus at once pervasive, powerful and fragile.

76.1.1 Pointers Are Not Addresses

In an attempt to rule out issues that come from direct addressing of memory with pointers, MISRA C states in Rule 11.4 that "*A conversion should not be performed between a pointer to object and an integer type*". As this rule is classified as only Advisory, MISRA C completes it with two Required rules:

- Rule 11.6: "*A cast shall not be performed between pointer to void and an arithmetic type*"
- Rule 11.7: "*A cast shall not be performed between pointer to object and a non-integer arithmetic type*"

In Ada, pointers are not addresses, and addresses are not integers. An opaque standard type `System.Address` is used for addresses, and conversions to/from integers are provided in a standard package `System.Storage_Elements`. The previous C code can be written as follows in Ada:

Listing 2: pointer.adb

```

1  with System;
2  with System.Storage_Elements;
3
4  procedure Pointer is
5    A : constant System.Address := System.Storage_Elements.To_Address (42);
6    M : aliased Integer with Address => A;
7    P : constant access Integer := M'Access;
8  begin
9    P.all := 0;
10 end Pointer;
```

The integer value 42 is converted to a memory address A by calling `System.Storage_Elements.To_Address`, which is then used as the address of integer variable M. The pointer variable P is set to point to M (which is allowed because M is declared as `aliased`).

Ada requires more verbiage than C:

- The integer value 42 must be explicitly converted to type `Address`
- To get a pointer to a declared variable such as M, the declaration must be marked as `aliased`

The added syntax helps first in making clear what is happening and, second, in ensuring that a potentially dangerous feature (assigning to a value at a specific machine address) is not used inadvertently.

The above example is legal in SPARK, but the SPARK analysis tool issues warnings as it cannot control how the program or its environment may update the memory cell at address 42.

76.1.2 Pointers Are Not References

Passing parameters by reference is critical for efficient programs, but the absence of references distinct from pointers in C incurs a serious risk. Any parameter of a pointer type can be copied freely to a variable whose lifetime is longer than the object pointed to, a problem known as "dangling pointers". MISRA C forbids such uses in Rule 18.6: "*The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist*". Unfortunately, enforcing this rule is difficult, as it is undecidable.

In SPARK, parameters can be passed by reference, but no pointer to the parameter can be stored past the return point of the function, which completely solves this issue. In fact, the

decision to pass a parameter by copy or by reference rests in many cases with the compiler, but such compiler dependency has no effect on the functional behavior of a SPARK program. In the example below, the compiler may decide to pass parameter P of procedure Rotate_X either by copy or by reference, but regardless of the choice the postcondition of Rotate_X will hold: the final value of P will be modified by rotation around the X axis.

Listing 3: geometry.ads

```

1 package Geometry is
2
3     type Point_3D is record
4         X, Y, Z : Float;
5     end record;
6
7     procedure Rotate_X (P : in out Point_3D) with
8         Post => P = P'Old'Update (Y => P.Z'Old, Z => -P.Y'Old);
9
10    end Geometry;

```

Listing 4: geometry.adb

```

1 package body Geometry is
2
3     procedure Rotate_X (P : in out Point_3D) is
4         Tmp : constant Float := P.Y;
5     begin
6         P.Y := P.Z;
7         P.Z := -Tmp;
8     end Rotate_X;
9
10    end Geometry;

```

Build output

```
geometry.ads:8:23: warning: attribute Update is an obsolescent feature [-gnatwj]
geometry.ads:8:23: warning: use a delta aggregate instead [-gnatwj]
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
geometry.ads:8:14: info: postcondition proved
geometry.ads:8:23: warning: attribute Update is an obsolescent feature [-gnatwj]
geometry.ads:8:23: warning: use a delta aggregate instead [-gnatwj]
```

SPARK's analysis tool can mathematically prove that the postcondition is true.

76.1.3 Pointers Are Not Arrays

The greatest source of vulnerabilities regarding pointers is their use as substitutes for arrays. Although the C language has a syntax for declaring and accessing arrays, this is just a thin syntactic layer on top of pointers. Thus:

- Array access is just pointer arithmetic;
- If a function is to manipulate an array then the array's length must be separately passed as a parameter; and
- The program is susceptible to the various vulnerabilities originating from the confusion of pointers and arrays, such as buffer overflow.

Consider a function that counts the number of times a value is present in an array. In C, this could be written:

Listing 5: main.c

```
1 #include <stdio.h>
2
3 int count(int *p, int len, int v) {
4     int count = 0;
5     while (len--) {
6         if (*p++ == v) {
7             count++;
8         }
9     }
10    return count;
11}
12
13 int main() {
14     int p[5] = {0, 3, 9, 3, 3};
15     int c = count(p, 5, 3);
16     printf("value 3 is seen %d times in p\n", c);
17     return 0;
18}
```

Runtime output

```
value 3 is seen 3 times in p
```

Function count has no control over the range of addresses accessed from pointer p. The critical property that the len parameter is a valid length for an array of integers pointed to by parameter p rests completely with the caller of count, and count has no way to check that this is true.

To mitigate the risks associated with pointers being used for arrays, MISRA C contains eight rules in a section on "Pointers and arrays". These rules forbid pointer arithmetic (Rule 18.4) or, if this Advisory rule is not followed, require pointer arithmetic to stay within bounds (Rule 18.1). But, even if we rewrite the loop in count to respect all decidable MISRA C rules, the program's correctness still depends on the caller of count passing a correct value of len:

Listing 6: main.c

```
1 #include <stdio.h>
2
3 int count(int *p, int len, int v) {
4     int count = 0;
5     for (int i = 0; i < len; i++) {
6         if (p[i] == v) {
7             count++;
8         }
9     }
10    return count;
11}
12
13 int main() {
14     int p[5] = {0, 3, 9, 3, 3};
15     int c = count(p, 5, 3);
16     printf("value 3 is seen %d times in p\n", c);
17     return 0;
18}
```

Runtime output

```
value 3 is seen 3 times in p
```

The resulting code is more readable, but still vulnerable to incorrect values of parameter `len` passed by the caller of `count`, which violates undecidable MISRA C Rules 18.1 (pointer arithmetic should stay within bounds) and 1.3 (no undefined behavior). Contrast this with the same function in SPARK (and Ada):

Listing 7: types.ads

```
1 package Types is
2     type Int_Array is array (Positive range <>) of Integer;
3 end Types;
```

Listing 8: count.ads

```
1 with Types; use Types;
2
3 function Count (P : Int_Array; V : Integer) return Natural;
```

Listing 9: count.adb

```
1 function Count (P : Int_Array; V : Integer) return Natural is
2     Count : Natural := 0;
3 begin
4     for I in P'Range loop
5         if P (I) = V then
6             Count := Count + 1;
7         end if;
8     end loop;
9     return Count;
10 end Count;
```

Listing 10: test_count.adb

```
1 with Ada.Text_Io; use Ada.Text_Io;
2 with Types; use Types;
3 with Count;
4
5 procedure Test_Count is
6     P : constant Int_Array := (0, 3, 9, 3, 3);
7     C : constant Integer := Count (P, 3);
8 begin
9     Put_Line ("value 3 is seen" & C'Img & " times in p");
10 end Test_Count;
```

Runtime output

```
value 3 is seen 3 times in p
```

The array parameter `P` is not simply a homogeneous sequence of `Integer` values. The compiler must represent `P` so that its lower and upper bounds (`P'First` and `P'Last`) and thus also its length (`P'Length`) can be retrieved. Function `Count` can simply loop over the range of valid array indexes `P'First .. P'Last` (or `P'Range` for short). As a result, function `Count` can be verified in isolation to be free of vulnerabilities such as buffer overflow, as it does not depend on the values of parameters passed by its callers. In fact, we can go further in SPARK and show that the value returned by `Count` is no greater than the length of parameter `P` by stating this property in the postcondition of `Count` and asking the SPARK analysis tool to prove it:

Listing 11: types.ads

```

1 package Types is
2     type Int_Array is array (Positive range <>) of Integer;
3 end Types;

```

Listing 12: count.ads

```

1 with Types; use Types;
2
3 function Count (P : Int_Array; V : Integer) return Natural with
4     Post => Count'Result <= P'Length;

```

Listing 13: count.adb

```

1 function Count (P : Int_Array; V : Integer) return Natural
2 is
3     Count : Natural := 0;
4 begin
5     for I in P'Range loop
6         pragma Loop_Invariant (Count <= I - P'First);
7         if P (I) = V then
8             Count := Count + 1;
9         end if;
10        end loop;
11        return Count;
12    end Count;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
count.adb:6:30: info: loop invariant preservation proved
count.adb:6:30: info: loop invariant initialization proved
count.adb:6:41: info: overflow check proved
count.adb:8:25: info: overflow check proved
count.ads:4:11: info: postcondition proved
count.ads:4:28: info: range check proved

```

The only help that SPARK analysis required from the programmer, in order to prove the postcondition, is a loop invariant (a special kind of assertion) that reflects the value of **Count** at each iteration.

76.1.4 Pointers Should Be Typed

The C language defines a special pointer type **void*** that corresponds to an untyped pointer. It is legal to convert any pointer type to and from **void***, which makes it a convenient way to simulate C++ style templates. Consider the following code which indirectly applies `assign_int` to integer `i` and `assign_float` to floating-point `f` by calling `assign` on both:

Listing 14: main.c

```

1 #include <stdio.h>
2
3 void assign_int (int *p) {
4     *p = 42;
5 }

```

(continues on next page)

(continued from previous page)

```

7 void assign_float (float *p) {
8     *p = 42.0;
9 }
10
11 typedef void (*assign_fun)(void *p);
12
13 void assign(assign_fun fun, void *p) {
14     fun(p);
15 }
16
17 int main() {
18     int i;
19     float f;
20     assign((assign_fun)&assign_int, &i);
21     assign((assign_fun)&assign_float, &f);
22     printf("i = %d; f = %f\n", i, f);
23 }
```

Runtime output

```
i = 42; f = 42.000000
```

The references to the variables `i` and `f` are implicitly converted to the `void*` type as a way to apply `assign` to any second parameter `p` whose type matches the argument type of its first argument `fun`. The use of an untyped argument means that the responsibility for the correct typing rests completely with the programmer. Swap `i` and `f` in the calls to `assign` and you still get a compilable program without warnings, that runs and produces completely bogus output:

```
i = 1109917696; f = 0.000000
```

instead of the expected:

```
i = 42; f = 42.000000
```

Generics in SPARK (and Ada) can implement the desired functionality in a fully typed way, with any errors caught at compile time, where procedure `Assign` applies its parameter procedure `Initialize` to its parameter `V`:

Listing 15: assign.ads

```

1 generic
2     type T is private;
3     with procedure Initialize (V : out T);
4 procedure Assign (V : out T);
```

Listing 16: assign.adb

```

1 procedure Assign (V : out T) is
2 begin
3     Initialize (V);
4 end Assign;
```

Listing 17: apply_assign.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Assign;
3
4 procedure Apply_Assign is
5     procedure Assign_Int (V : out Integer) is
```

(continues on next page)

(continued from previous page)

```

6   begin
7     V := 42;
8   end Assign_Int;

9
10  procedure Assign_Float (V : out Float) is
11    begin
12      V := 42.0;
13    end Assign_Float;

14
15  procedure Assign_I is new Assign (Integer, Assign_Int);
16  procedure Assign_F is new Assign (Float, Assign_Float);

17
18  I : Integer;
19  F : Float;
20  begin
21    Assign_I (I);
22    Assign_F (F);
23    Put_Line ("I =" & I'Img & "; F =" & F'Img);
24  end Apply_Assign;

```

Runtime output

```
I = 42; F = 4.20000E+01
```

The generic procedure `Assign` must be instantiated with a specific type for `T` and a specific procedure (taking a single `out` parameter of this type) for `Initialize`. The procedure resulting from the instantiation applies to a variable of this type. So switching `I` and `F` above would result in an error detected by the compiler. Likewise, an instantiation such as the following would also be a compile-time error:

```
procedure Assign_I is new Assign (Integer, Assign_Float);
```

76.2 Enforcing Strong Typing for Scalars

In C, all scalar types can be converted both implicitly and explicitly to any other scalar type. The semantics is defined by rules of *promotion* and *conversion*, which can confuse even experts. One example was noted earlier, in the *Preface* (page 869). Another example appears in an article introducing a safe library for manipulating scalars¹¹⁷ by Microsoft expert David LeBlanc. In its conclusion, the author acknowledges the inherent difficulty in understanding scalar type conversions in C, by showing an early buggy version of the code to produce the minimum signed integer:

```
return (T)(1 << (BitCount()-1));
```

The issue here is that the literal `1` on the left-hand side of the shift is an `int`, so on a 64-bit machine with 32-bit `int` and 64-bit type `T`, the above is shifting 32-bit value `1` by 63 bits. This is a case of undefined behavior, producing an unexpected output with the Microsoft compiler. The correction is to convert the first literal `1` to `T` before the shift:

```
return (T)((T)1 << (BitCount()-1));
```

Although he'd asked some expert programmers to review the code, no one found this problem.

To avoid these issues as much as possible, MISRA C defines its own type system on top of C types, in the section on "The essential type model" (eight rules). These can be seen as

¹¹⁷ <https://msdn.microsoft.com/en-us/library/ms972705.aspx>

additional typing rules, since all rules in this section are decidable, and can be enforced at the level of a single translation unit. These rules forbid in particular the confusing cases mentioned above. They can be divided into three sets of rules:

- restricting operations on types
- restricting explicit conversions
- restricting implicit conversions

76.2.1 Restricting Operations on Types

Apart from the application of some operations to floating-point arguments (the bitwise, mod and array access operations) which are invalid and reported by the compiler, all operations apply to all scalar types in C. MISRA C Rule 10.1 constrains the types on which each operation is possible as follows.

Arithmetic Operations on Arithmetic Types

Adding two Boolean values, or an Apple and an Orange, might sound like a bad idea, but it is easily done in C:

Listing 18: main.c

```

1 #include <stdbool.h>
2 #include <stdio.h>
3
4 int main() {
5     bool b1 = true;
6     bool b2 = false;
7     bool b3 = b1 + b2;
8
9     typedef enum {Apple, Orange} fruit;
10    fruit f1 = Apple;
11    fruit f2 = Orange;
12    fruit f3 = f1 + f2;
13
14    printf("b3 = %d; f3 = %d\n", b3, f3);
15
16    return 0;
17 }
```

Runtime output

```
b3 = 1; f3 = 1
```

No error from the compiler here. In fact, there is no undefined behavior in the above code. Variables b3 and f3 both end up with value 1. Of course it makes no sense to add Boolean or enumerated values, and thus MISRA C Rule 18.1 forbids the use of all arithmetic operations on Boolean and enumerated values, while also forbidding most arithmetic operations on characters. That leaves the use of arithmetic operations for signed or unsigned integers as well as floating-point types and the use of modulo operation % for signed or unsigned integers.

Here's an attempt to simulate the above C code in SPARK (and Ada):

Listing 19: bad_arith.ads

```
1 package Bad_Arith is
2
3     B1 : constant Boolean := True;
4     B2 : constant Boolean := False;
5     B3 : constant Boolean := B1 + B2;
6
7     type Fruit is (Apple, Orange);
8     F1 : constant Fruit := Apple;
9     F2 : constant Fruit := Orange;
10    F3 : constant Fruit := F1 + F2;
11
12 end Bad_Arith;
```

Build output

```
bad_arith.ads:5:32: error: there is no applicable operator "+" for type "Standard.
     Boolean"
bad_arith.ads:10:30: error: there is no applicable operator "+" for type "Fruit" de
     fined at line 7
gprbuild: *** compilation phase failed
```

It is possible, however, to get the predecessor of a Boolean or enumerated value with Value'Pred and its successor with Value'Succ, as well as to iterate over all values of the type:

Listing 20: ok_arith.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Ok_Arith is
4
5     B1 : constant Boolean := False;
6     B2 : constant Boolean := Boolean'Succ (B1);
7     B3 : constant Boolean := Boolean'Pred (B2);
8
9     type Fruit is (Apple, Orange);
10    F1 : constant Fruit := Apple;
11    F2 : constant Fruit := Fruit'Succ (F1);
12    F3 : constant Fruit := Fruit'Pred (F2);
13
14 begin
15    pragma Assert (B1 = B3);
16    pragma Assert (F1 = F3);
17
18    for B in Boolean loop
19        Put_Line (B'Img);
20    end loop;
21
22    for F in Fruit loop
23        Put_Line (F'Img);
24    end loop;
25 end Ok_Arith;
```

Runtime output

```
FALSE
TRUE
APPLE
ORANGE
```

Boolean Operations on Boolean

"Two bee or not two bee? Let's C":

Listing 21: main.c

```

1 #include <stdbool.h>
2 #include <stdio.h>
3
4 int main() {
5     typedef enum {Ape, Bee, Cat} Animal;
6     bool answer = (2 * Bee) || ! (2 * Bee);
7     printf("two bee or not two bee? %d\n", answer);
8     return 0;
9 }
```

Runtime output

```
two bee or not two bee? 1
```

The answer to the question posed by Shakespeare's Hamlet is 1, since it reduces to A **or** **not** A and this is true in classical logic.

As previously noted, MISRA C forbids the use of the multiplication operator with an operand of an enumerated type. Rule 18.1 also forbids the use of Boolean operations "and", "or", and "not" (&&, ||, !, respectively, in C) on anything other than Boolean operands. It would thus prohibit the Shakespearian code above.

Below is an attempt to express the same code in SPARK (and Ada), where the Boolean operators are **and**, **or**, and **not**. The **and** and **or** operators evaluate both operands, and the language also supplies short-circuit forms that evaluate the left operand and only evaluate the right operand when its value may affect the result.

Listing 22: bad_hamlet.ads

```

1 package Bad_Hamlet is
2     type Animal is (Ape, Bee, Cat);
3     Answer : Boolean := 2 * Bee or not 2 * Bee; -- Illegal
4 end Bad_Hamlet;
```

Build output

```
bad_hamlet.ads:3:28: error: expected type universal integer
bad_hamlet.ads:3:28: error: found type "Animal" defined at line 2
bad_hamlet.ads:3:43: error: expected a modular type
bad_hamlet.ads:3:43: error: found type "Animal" defined at line 2
gprbuild: *** compilation phase failed
```

As expected, the compiler rejects this code. There is no available ***** operation that works on an enumeration type, and likewise no available **or** or **not** operation.

Bitwise Operations on Unsigned Integers

Here's a genetic engineering example that combines a Bee with a Dog to produce a Cat, by manipulating the atomic structure (the bits in its representation):

Listing 23: main.c

```
1 #include <stdbool.h>
2 #include <assert.h>
3
4 int main() {
5     typedef enum {Ape, Bee, Cat, Dog} Animal;
6     Animal mutant = Bee ^ Dog;
7     assert (mutant == Cat);
8     return 0;
9 }
```

This algorithm works by accessing the underlying bitwise representation of Bee and Dog (0x01 and 0x03, respectively) and, by applying the exclusive-or operator `^`, transforming it into the underlying bitwise representation of a Cat (0x02). While powerful, manipulating the bits in the representation of values is best reserved for unsigned integers as illustrated in the book [Hacker's Delight](#)¹¹⁸. MISRA C Rule 18.1 thus forbids the use of all bitwise operations on anything but unsigned integers.

Below is an attempt to do the same in SPARK (and Ada). The bitwise operators are `and`, `or`, `xor`, and `not`, and the related bitwise functions are `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left` and `Rotate_Right`:

Listing 24: bad_genetics.ads

```
1 package Bad_Genetics is
2     type Animal is (Ape, Bee, Cat, Dog);
3     Mutant : Animal := Bee xor Dog;    -- ERROR
4     pragma Assert (Mutant = Cat);
5 end Bad_Genetics;
```

Build output

```
bad_genetics.ads:3:27: error: there is no applicable operator "Xor" for type
  ↵ "Animal" defined at line 2
gprbuild: *** compilation phase failed
```

The declaration of `Mutant` is illegal, since the `xor` operator is only available for Boolean and unsigned integer (modular) values; it is not available for `Animal`. The same restriction applies to the other bitwise operators listed above. If we really wanted to achieve the effect of the above code in legal SPARK (or Ada), then the following approach will work (the type `Unsigned_8` is an 8-bit modular type declared in the predefined package `Interfaces`).

Listing 25: unethical_genetics.ads

```
1 with Interfaces; use Interfaces;
2 package Unethical_Genetics is
3     type Animal is (Ape, Bee, Cat, Dog);
4     A : constant array (Animal) of Unsigned_8 :=
5         (Animal'Pos (Ape), Animal'Pos (Bee),
6          Animal'Pos (Cat), Animal'Pos (Dog));
7     Mutant : Animal := Animal'Val (A (Bee) xor A (Dog));
8     pragma Assert (Mutant = Cat);
9 end Unethical_Genetics;
```

Build output

¹¹⁸ <http://www.hackersdelight.org/>

```
unethical_genetics.ads:8:26: warning: condition can only be False if invalid
  ↵values present [-gnatwc]
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
```

Note that **and**, **or**, **not** and **xor** are used both as logical operators and as bitwise operators, but there is no possible confusion between these two uses. Indeed the use of such operators on values from modular types is a natural generalization of their uses on Boolean, since values from modular types are often interpreted as arrays of Booleans.

76.2.2 Restricting Explicit Conversions

A simple way to bypass the restrictions of Rule 10.1 is to explicitly convert the arguments of an operation to a type that the rule allows. While it can often be useful to cast a value from one type to another, many casts that are allowed in C are either downright errors or poor replacements for clearer syntax.

One example is to cast from a scalar type to Boolean. A better way to express `(bool)x` is to compare `x` to the zero value of its type: `x != 0` for integers, `x != 0.0` for floats, `x != '0'` for characters, `x != Enum` where `Enum` is the first enumerated value of the type. Thus, MISRA C Rule 10.5 advises avoiding casting non-Boolean values to Boolean.

Rule 10.5 also advises avoiding other casts that are, at best, obscure:

- from a Boolean to any other scalar type
- from a floating-point value to an enumeration or a character
- from any scalar type to an enumeration

The rules are not symmetric, so although a float should not be cast to an enum, casting an enum to a float is allowed. Similarly, although it is advised to not cast a character to an enum, casting an enum to a character is allowed.

The rules in SPARK are simpler. There are no conversions between numeric types (integers, fixed-point and floating-point) and non-numeric types (such as Boolean, Character, and other enumeration types). Conversions between different non-numeric types are limited to those that make semantic sense, for example between a derived type and its parent type. Any numeric type can be converted to any other numeric type, with precise rules for rounding/truncating values when needed and run-time checking that the converted value is in the range associated with the target type.

76.2.3 Restricting Implicit Conversions

Rules 10.1 and 10.5 restrict operations on types and explicit conversions. That's not enough to avoid problematic C programs; a program violating one of these rules can be expressed using only implicit type conversions. For example, the Shakespearian code in section *Boolean Operations on Boolean* (page 899) can be reformulated to satisfy both Rules 10.1 and 10.5:

Listing 26: main.c

```
1 #include <stdbool.h>
2 #include <stdio.h>
3
4 int main() {
```

(continues on next page)

(continued from previous page)

```

5   typedef enum {Ape, Bee, Cat} Animal;
6   int b = Bee;
7   bool t = 2 * b;
8   bool answer = t || !t;
9   printf("two bee or not two bee? %d\n", answer);
10  return 0;
11 }
```

Runtime output

```
two bee or not two bee? 1
```

Here, we're implicitly converting the enumerated value Bee to an int, and then implicitly converting the integer value `2 * b` to a Boolean. This does not violate 10.1 or 10.5, but it is prohibited by MISRA C Rule 10.3: "*The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category*".

Rule 10.1 also does not prevent arguments of an operation from being inconsistent, for example comparing a floating-point value and an enumerated value. But MISRA C Rule 10.4 handles this situation: "*Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category*".

In addition, three rules in the "Composite operators and expressions" section avoid common mistakes related to the combination of explicit/implicit conversions and operations.

The rules in SPARK (and Ada) are far simpler: there are no implicit conversions! This applies both between types of a different *essential type category* as MISRA C puts it, as well as between types that are structurally the same but declared as different types.

Listing 27: bad_conversions.adb

```

1  procedure Bad_Conversions is
2    pragma Warnings (Off);
3    F : Float := 0.0;
4    I : Integer := 0;
5    type Animal is (Ape, Bee, Cat);
6    type My_Animal is new Animal; -- derived type
7    A : Animal := Cat;
8    M : My_Animal := Bee;
9    B : Boolean := True;
10   C : Character := 'a';
11 begin
12   F := I;    -- ERROR
13   I := A;    -- ERROR
14   A := B;    -- ERROR
15   M := A;    -- ERROR
16   B := C;    -- ERROR
17   C := F;    -- ERROR
18 end Bad_Conversions;
```

Build output

```
bad_conversions.adb:12:09: error: expected type "Standard.Float"
bad_conversions.adb:12:09: error: found type "Standard.Integer"
bad_conversions.adb:13:09: error: expected type "Standard.Integer"
bad_conversions.adb:13:09: error: found type "Animal" defined at line 5
bad_conversions.adb:14:09: error: expected type "Animal" defined at line 5
bad_conversions.adb:14:09: error: found type "Standard.Boolean"
bad_conversions.adb:15:09: error: expected type "My_Animal" defined at line 6
bad_conversions.adb:15:09: error: found type "Animal" defined at line 5
bad_conversions.adb:16:09: error: expected type "Standard.Boolean"
```

(continues on next page)

(continued from previous page)

```
bad_conversions.adb:16:09: error: found type "Standard.Character"
bad_conversions.adb:17:09: error: expected type "Standard.Character"
bad_conversions.adb:17:09: error: found type "Standard.Float"
gprbuild: *** compilation phase failed
```

The compiler reports a mismatch on every statement in the above procedure (the declarations are all legal).

Adding explicit conversions makes the assignments to F and M valid, since SPARK (and Ada) allow conversions between numeric types and between a derived type and its parent type, but all other conversions are illegal:

Listing 28: bad_conversions.adb

```
1  procedure Bad_Conversions is
2    pragma Warnings (Off);
3    F : Float := 0.0;
4    I : Integer := 0;
5    type Animal is (Ape, Bee, Cat);
6    type My_Animal is new Animal; -- derived type
7    A : Animal := Cat;
8    M : My_Animal := Bee;
9    B : Boolean := True;
10   C : Character := 'a';
11 begin
12   F := Float (I);      -- OK
13   I := Integer (A);   -- ERROR
14   A := Animal (B);    -- ERROR
15   M := My_Animal (A); -- OK
16   B := Boolean (C);   -- ERROR
17   C := Character (F); -- ERROR
18 end Bad_Conversions;
```

Build output

```
bad_conversions.adb:13:18: error: illegal operand for numeric conversion
bad_conversions.adb:14:09: error: invalid conversion, not compatible with type
  ↵ "Standard.Boolean"
bad_conversions.adb:16:09: error: invalid conversion, not compatible with type
  ↵ "Standard.Character"
bad_conversions.adb:17:09: error: invalid conversion, not compatible with type
  ↵ "Standard.Float"
gprbuild: *** compilation phase failed
```

Although an enumeration value cannot be converted to an integer (or *vice versa*) either implicitly or explicitly, SPARK (and Ada) provide functions to obtain the effect of a type conversion. For any enumeration type T, the function T'Pos(e) takes an enumeration value from type T and returns its relative position as an integer, starting at 0. For example, Animal'Pos(Bee) is 1, and Boolean'Pos(False) is 0. In the other direction, T'Val(n), where n is an integer, returns the enumeration value in type T at relative position n. If n is negative or greater than T'Pos(T'Last) then a run-time exception is raised.

Hence, the following is valid SPARK (and Ada) code; **Character** is defined as an enumeration type:

Listing 29: ok_conversions.adb

```
1  procedure Ok_Conversions is
2    pragma Warnings (Off);
3    F : Float := 0.0;
4    I : Integer := 0;
```

(continues on next page)

(continued from previous page)

```
5  type Animal is (Ape, Bee, Cat);
6  type My_Animal is new Animal;
7  A : Animal := Cat;
8  M : My_Animal := Bee;
9  B : Boolean := True;
10 C : Character := 'a';
11 begin
12   F := Float (I);
13   I := Animal'Pos (A);
14   I := My_Animal'Pos (M);
15   I := Boolean'Pos (B);
16   I := Character'Pos (C);
17   I := Integer (F);
18   A := Animal'Val (2);
19 end Ok_Conversions;
```

CHAPTER
SEVENTYSEVEN

INITIALIZING DATA BEFORE USE

As with most programming languages, C does not require that variables be initialized at their declaration, which makes it possible to unintentionally read uninitialized data. This is a case of undefined behavior, which can sometimes be used to attack the program.

77.1 Detecting Reads of Uninitialized Data

MISRA C attempts to prevent reads of uninitialized data in a specific section on "Initialization", containing five rules. The most important is Rule 9.1: "*The value of an object with automatic storage duration shall not be read before it has been set*". The first example in the rule is interesting, as it shows a non-trivial (and common) case of conditional initialization, where a function f initializes an output parameter p only in some cases, and the caller g of f ends up reading the value of the variable u passed in argument to f in cases where it has not been initialized:

Listing 1: f.h

```
1 #include <stdint.h>
2
3 void f ( int b, uint16_t *p );
```

Listing 2: f.c

```
1 #include "f.h"
2
3 void f ( int b, uint16_t *p )
4 {
5     if ( b )
6     {
7         *p = 3U;
8     }
9 }
```

Listing 3: g.c

```
1 #include <stdint.h>
2 #include "f.h"
3
4 static void g (void)
5 {
6     uint16_t u;
7
8     f ( 0, &u );
9
10    if ( u == 3U )
```

(continues on next page)

(continued from previous page)

```

11  {
12    /* Non-compliant use - "u" has not been assigned a value. */
13  }
14 }
```

Detecting the violation of Rule 9.1 can be arbitrarily complex, as the program points corresponding to a variable's initialization and read can be separated by many calls and conditions. This is one of the undecidable rules, for which most MISRA C checkers won't detect all violations.

In SPARK, the guarantee that all reads are to initialized data is enforced by the SPARK analysis tool, GNATprove, through what is referred to as *flow analysis*. Every subprogram is analyzed separately to check that it cannot read uninitialized data. To make this modular analysis possible, SPARK programs need to respect the following constraints:

- all inputs of a subprogram should be initialized on subprogram entry
- all outputs of a subprogram should be initialized on subprogram return

Hence, given the following code translated from C, GNATprove reports that function F might not always initialize output parameter P:

Listing 4: init.ads

```

1 with Interfaces; use Interfaces;
2
3 package Init is
4   procedure F (B : Boolean; P : out Unsigned_16);
5   procedure G;
6 end Init;
```

Listing 5: init.adb

```

1 package body Init is
2
3   procedure F (B : Boolean; P : out Unsigned_16) is
4     begin
5       if B then
6         P := 3;
7       end if;
8     end F;
9
10  procedure G is
11    U : Unsigned_16;
12  begin
13    F (False, U);
14
15    if U = 3 then
16      null;
17    end if;
18  end G;
19
20 end Init;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init.adb:15:07: warning: if statement has no effect [-gnatwr]
init.ads:4:30: medium: "P" might not be initialized in "F" [reason for check: OUT_U
↳parameter should be initialized on return] [possible fix: initialize "P" on all_U
```

(continues on next page)

(continued from previous page)

```
→paths or make "P" an IN OUT parameter]
gnatprove: unproved check messages considered as errors
```

We can correct the program by initializing P to value 0 when condition B is not satisfied:

Listing 6: init.ads

```
1 with Interfaces; use Interfaces;
2
3 package Init is
4     procedure F (B : Boolean; P : out Unsigned_16);
5     procedure G;
6 end Init;
```

Listing 7: init.adb

```
1 package body Init is
2
3     procedure F (B : Boolean; P : out Unsigned_16) is
4     begin
5         if B then
6             P := 3;
7         else
8             P := 0;
9         end if;
10    end F;
11
12    procedure G is
13        U : Unsigned_16;
14    begin
15        F (False, U);
16
17        if U = 3 then
18            null;
19        end if;
20    end G;
21
22 end Init;
```

Build output

```
init.adb:17:07: warning: if statement has no effect [-gnatwr]
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init.adb:13:07: info: initialization of "U" proved
init.adb:17:07: warning: if statement has no effect [-gnatwr]
init.ads:4:30: info: initialization of "P" proved
```

GNATprove now does not report any possible reads of uninitialized data. On the contrary, it confirms that all reads are made from initialized data.

In contrast with C, SPARK does not guarantee that global data (called *library-level* data in SPARK and Ada) is zero-initialized at program startup. Instead, GNATprove checks that all global data is explicitly initialized (at declaration or elsewhere) before it is read. Hence it goes beyond the MISRA C Rule 9.1, which considers global data as always initialized even if the default value of all-zeros might not be valid data for the application. Here's a variation of the above code where variable U is now global:

Listing 8: init.ads

```

1  with Interfaces; use Interfaces;
2
3  package Init is
4      U : Unsigned_16;
5      procedure F (B : Boolean);
6      procedure G;
7  end Init;

```

Listing 9: init.adb

```

1  package body Init is
2
3      procedure F (B : Boolean) is
4          begin
5              if B then
6                  U := 3;
7              end if;
8          end F;
9
10     procedure G is
11         begin
12             F (False);
13
14             if U = 3 then
15                 null;
16             end if;
17         end G;
18
19 end Init;

```

Listing 10: call_init.adb

```

1  with Init;
2
3  procedure Call_Init is
4      begin
5          Init.G;
6      end Call_Init;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
call_init.adb:5:08: medium: "U" might not be initialized after elaboration of main
  ↪program "Call_Init"
init.adb:14:07: warning: if statement has no effect [-gnatwr]
init.adb:14:07: warning: statement has no effect
gnatprove: unproved check messages considered as errors

```

GNATprove reports here that variable U might not be initialized at program startup, which is indeed the case here. It reports this issue on the main program Call_Init because its analysis showed that F needs to take U as an initialized input (since F is not initializing U on all paths, U keeps its value on the other path, which needs to be an initialized value), which means that G which calls F also needs to take U as an initialized input, which in turn means that Call_Init which calls G also needs to take U as an initialized input. At this point, we've reached the main program, so the initialization phase (referred to as *elaboration* in SPARK and Ada) should have taken care of initializing U. This is not the case here, hence the message from GNATprove.

It is possible in SPARK to specify that G should initialize variable U; this is done with a *data dependency* contract introduced with aspect Global following the declaration of procedure G:

Listing 11: init.ads

```

1 with Interfaces; use Interfaces;
2
3 package Init is
4   U : Unsigned_16;
5   procedure F (B : Boolean);
6   procedure G with Global => (Output => U);
7 end Init;
```

Listing 12: init.adb

```

1 package body Init is
2
3   procedure F (B : Boolean) is
4   begin
5     if B then
6       U := 3;
7     end if;
8   end F;
9
10  procedure G is
11  begin
12    F (False);
13
14    if U = 3 then
15      null;
16    end if;
17  end G;
18
19 end Init;
```

Listing 13: call_init.adb

```

1 with Init;
2
3 procedure Call_Init is
4 begin
5   Init.G;
6 end Call_Init;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init.adb:12:07: high: "U" is not initialized
init.adb:12:07: high: "U" is not an input in the Global contract of subprogram "G" ↴
  ↪at init.ads:6
init.adb:12:07: high: either make "U" an input in the Global contract or ↴
  ↪initialize it before use
init.adb:14:07: warning: if statement has no effect [-gnatwr]
init.adb:14:07: warning: statement has no effect
gnatprove: unproved check messages considered as errors
```

GNATprove reports the error on the call to F in G, as it knows at this point that F needs U to be initialized but the calling context in G cannot provide that guarantee. If we provide the same data dependency contract for F, then GNATprove reports the error on F itself, similarly to what we saw for an output parameter U.

77.2 Detecting Partial or Redundant Initialization of Arrays and Structures

The other rules in the section on "Initialization" deal with common errors in initializing aggregates and *designated initializers* in C99 to initialize a structure or array at declaration. These rules attempt to patch holes created by the lax syntax and rules in C standard. For example, here are five valid initializations of an array of 10 elements in C:

Listing 14: main.c

```

1 int main() {
2     int a[10] = {0};
3     int b[10] = {0, 0};
4     int c[10] = {0, [8] = 0};
5     int d[10] = {0, [8] = 0, 0};
6     int e[10] = {0, [8] = 0, 0, [8] = 1};
7     return 0;
8 }
```

Only a is fully initialized to all-zeros in the above code snippet. MISRA C Rule 9.3 thus forbids all other declarations by stating that "*Arrays shall not be partially initialized*". In addition, MISRA C Rule 9.4 forbids the declaration of e by stating that "*An element of an object shall not be initialised more than once*" (in e's declaration, the element at index 8 is initialized twice).

The same holds for initialization of structures. Here is an equivalent set of declarations with the same potential issues:

Listing 15: main.c

```

1 int main() {
2     typedef struct { int x; int y; int z; } rec;
3     rec a = {0};
4     rec b = {0, 0};
5     rec c = {0, .y = 0};
6     rec d = {0, .y = 0, 0};
7     rec e = {0, .y = 0, 0, .y = 1};
8     return 0;
9 }
```

Here only a, d and e are fully initialized. MISRA C Rule 9.3 thus forbids the declarations of b and c. In addition, MISRA C Rule 9.4 forbids the declaration of e.

In SPARK and Ada, the aggregate used to initialize an array or a record must fully cover the components of the array or record. Violations lead to compilation errors, both for records:

Listing 16: init_record.ads

```

1 package Init_Record is
2     type Rec is record
3         X, Y, Z : Integer;
4     end record;
5     R : Rec := (X => 1); -- ERROR, Y and Z not specified
6 end Init_Record;
```

Build output

```

init_record.ads:5:15: error: no value supplied for component "Y"
init_record.ads:5:15: error: no value supplied for component "Z"
gprbuild: *** compilation phase failed
```

and for arrays:

Listing 17: init_array.ads

```

1 package Init_Array is
2   type Arr is array (1 .. 10) of Integer;
3   A : Arr := (1 => 1); -- ERROR, elements 2..10 not specified
4 end Init_Array;
```

Build output

```

init_array.ads:3:15: warning: too few elements for type "Arr" defined at line 2
  ↵[enabled by default]
init_array.ads:3:15: warning: expected 10 elements; found 1 element [enabled by
  ↵default]
init_array.ads:3:15: warning: Constraint_Error will be raised at run time [enabled
  ↵by default]
```

Similarly, redundant initialization leads to compilation errors for records:

Listing 18: init_record.ads

```

1 package Init_Record is
2   type Rec is record
3     X, Y, Z : Integer;
4   end record;
5   R : Rec := (X => 1, Y => 1, Z => 1, X => 2); -- ERROR, X duplicated
6 end Init_Record;
```

Build output

```

init_record.ads:5:40: error: more than one value supplied for "X"
gprbuild: *** compilation phase failed
```

and for arrays:

Listing 19: init_array.ads

```

1 package Init_Array is
2   type Arr is array (1 .. 10) of Integer;
3   A : Arr := (1 .. 8 => 1, 9 .. 10 => 2, 7 => 3); -- ERROR, A(7) duplicated
4 end Init_Array;
```

Build output

```

init_array.ads:3:43: error: index value in array aggregate duplicates the one
  ↵given at line 3
init_array.ads:3:43: error:    7
gprbuild: *** compilation phase failed
```

Finally, while it is legal in Ada to leave uninitialized parts in a record or array aggregate by using the box notation (meaning that the default initialization of the type is used, which may be no initialization at all), SPARK analysis rejects such use when it leads to components not being initialized, both for records:

Listing 20: init_record.ads

```

1 package Init_Record is
2   type Rec is record
3     X, Y, Z : Integer;
4   end record;
5   R : Rec := (X => 1, others => <>); -- ERROR, Y and Z not specified
6 end Init_Record;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init_record.ads:5:04: error: "R" is not allowed in SPARK (due to box notation,
  ↪without default initialization)
init_record.ads:5:04: error: violation of pragma SPARK_Mode at /vagrant/frontend/
  ↪dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_
  ↪Record_3/main.adc:12
init_record.ads:5:15: error: box notation without default initialization is not
  ↪allowed in SPARK (SPARK RM 4.3(1))
init_record.ads:5:15: error: violation of pragma SPARK_Mode at /vagrant/frontend/
  ↪dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_
  ↪Record_3/main.adc:12
gnatprove: error during analysis of data and information flow
```

and for arrays:

Listing 21: init_array.ads

```
1 package Init_Array is
2   type Arr is array (1 .. 10) of Integer;
3   A : Arr := (1 .. 8 => 1, 9 .. 10 => <>);  --  ERROR, A(9..10) not specified
4 end Init_Array;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
init_array.ads:3:37: error: box notation without default initialization is not
  ↪allowed in SPARK (SPARK RM 4.3(1))
init_array.ads:3:37: error: violation of pragma SPARK_Mode at /vagrant/frontend/
  ↪dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Initialization/Init_
  ↪Array_3/main.adc:12
gnatprove: error during analysis of data and information flow
```

CHAPTER
SEVENTYEIGHT

CONTROLLING SIDE EFFECTS

As with most programming languages, C allows side effects in expressions. This leads to subtle issues about conflicting side effects, when subexpressions of the same expression read/write the same variable.

78.1 Preventing Undefined Behavior

Conflicting side effects are a kind of undefined behavior; the C Standard (section 6.5) defines the concept as follows:

"Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored"

This legalistic wording is somewhat opaque, but the notion of sequence points is summarized in Annex C of the C90 and C99 standards. MISRA C repeats these conditions in the Amplification of Rule 13.2, including the read of a volatile variable as a side effect similar to writing a variable.

This rule is undecidable, so MISRA C completes it with two rules that provide simpler restrictions preventing some side effects in expressions, thus reducing the potential for undefined behavior:

- Rule 13.3: *"A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator".*
- Rule 13.4: *"The result of an assignment operator should not be used".*

In practice, conflicting side effects usually manifest themselves as portability issues, since the result of the evaluation of an expression depends on the order in which a compiler decides to evaluate its subexpressions. So changing the compiler version or the target platform might lead to a different behavior of the application.

To reduce the dependency on evaluation order, MISRA C Rule 13.1 states: *"Initializer lists shall not contain persistent side effects"*. This case is theoretically different from the previously mentioned conflicting side effects, because initializers that comprise an initializer list are separated by sequence points, so there is no risk of undefined behavior if two initializers have conflicting side effects. But given that initializers are executed in an unspecified order, the result of a conflict is potentially as damaging for the application.

78.2 Reducing Programmer Confusion

Even in cases with no undefined or unspecified behavior, expressions with multiple side effects can be confusing to programmers reading or maintaining the code. This problem arises in particular with C's increment and decrement operators that can be applied prior to or after the expression evaluation, and with the assignment operator = in C since it can easily be mistaken for equality. Thus MISRA C forbids the use of the increment / decrement (Rule 13.3) and assignment (Rule 13.4) operators in expressions that have other potential side effects.

In other cases, the presence of expressions with side effects might be confusing, if the programmer wrongly thinks that the side effects are guaranteed to occur. Consider the function decrease_until_one_is_null below, which decreases both arguments until one is null:

Listing 1: main.c

```
1 #include <stdio.h>
2
3 void decrease_until_one_is_null (int *x, int *y) {
4     if (x == 0 || y == 0) {
5         return;
6     }
7     while (--*x != 0 && --*y != 0) {
8         // nothing
9     }
10 }
11
12 int main() {
13     int x = 42, y = 42;
14     decrease_until_one_is_null (&x, &y);
15     printf("x = %d, y = %d\n", x, y);
16     return 0;
17 }
```

Runtime output

```
x = 0, y = 1
```

The program produces the following output:

```
x = 0, y = 1
```

I.e., starting from the same value 42 for both x and y, only x has reached the value zero after decrease_until_one_is_null returns. The reason is that the side effect on y is performed only conditionally. To avoid such surprises, MISRA C Rule 13.5 states: "*The right hand operand of a logical && or || operator shall not contain persistent side effects*"; this rule forbids the code above.

MISRA C Rule 13.6 similarly states: "*The operand of the sizeof operator shall not contain any expression which has potential side effects*". Indeed, the operand of **sizeof** is evaluated only in rare situations, and only according to C99 rules, which makes any side effect in such an operand a likely mistake.

78.3 Side Effects and SPARK

In SPARK, expressions cannot have side effects; only statements can. In particular, there are no increment/decrement operators, and no assignment operator. There is instead an assignment statement, whose syntax using `:=` clearly distinguishes it from equality (using `=`). And in any event an expression is not allowed as a statement and thus a construct such as `X = Y;` would be illegal. Here is how a variable `X` can be assigned, incremented and decremented:

```
X := 1;
X := X + 1;
X := X - 1;
```

There are two possible side effects when evaluating an expression:

- a read of a volatile variable
- a side effect occurring inside a function that the expression calls

Reads of volatile variables in SPARK are restricted to appear immediately at statement level, so the following is not allowed:

Listing 2: volatile_read.ads

```
1 package Volatile_Read is
2   X : Integer with Volatile;
3   procedure P (Y : out Integer);
4 end Volatile_Read;
```

Listing 3: volatile_read.adb

```
1 package body Volatile_Read is
2   procedure P (Y : out Integer) is
3     begin
4       Y := X - X; -- ERROR
5     end P;
6   end Volatile_Read;
```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
volatile_read.adb:4:12: error: volatile object cannot appear in this context
  ↳(SPARK RM 7.1.3(10))
volatile_read.adb:4:16: error: volatile object cannot appear in this context
  ↳(SPARK RM 7.1.3(10))
gnatprove: error during generation of Global contracts
```

Instead, every read of a volatile variable must occur immediately before being assigned to another variable, as follows:

Listing 4: volatile_read.ads

```
1 package Volatile_Read is
2   X : Integer with Volatile;
3   procedure P (Y : out Integer);
4 end Volatile_Read;
```

Listing 5: volatile_read.adb

```
1 package body Volatile_Read is
2   procedure P (Y : out Integer) is
```

(continues on next page)

(continued from previous page)

```

3   X1 : constant Integer := X;
4   X2 : constant Integer := X;
5 begin
6   Y := X1 - X2;
7 end P;
8 end Volatile_Read;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
volatile_read.ads:3:17: info: initialization of "Y" proved

```

Note here that the order of capture of the volatile value of X might be significant. For example, X might denote a quantity which only increases, like clock time, so that the above expression $X_1 - X_2$ would always be negative or zero.

Even more significantly, functions in SPARK cannot have side effects; only procedures can. The only effect of a SPARK function is the computation of a result from its inputs, which may be passed as parameters or as global variables. In particular, SPARK functions cannot have **out** or **in out** parameters:

Listing 6: bad_function.ads

```

1 function Bad_Function (X, Y : Integer; Sum, Max : out Integer) return Boolean;
2 -- ERROR, since "out" parameters are not allowed

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
bad_function.ads:1:10: error: function with "out" parameter is not allowed in SPARK
bad_function.ads:1:10: error: violation of pragma SPARK_Mode at /vagrant/frontend/
  ↪dist/test_output/projects/Courses/SPARK_For_The_MISRA_C_Dev/Side_Effect/Function_
  ↪With_Out_Param/main.adc:12
gnatprove: error during analysis of data and information flow

```

More generally, SPARK does not allow functions that have a side effect in addition to returning their result, as is typical of many idioms in other languages, for example when setting a new value and returning the previous one:

Listing 7: bad_functions.ads

```

1 package Bad_Functions is
2   function Set (V : Integer) return Integer;
3   function Get return Integer;
4 end Bad_Functions;

```

Listing 8: bad_functions.adb

```

1 package body Bad_Functions is
2
3   Value : Integer := 0;
4
5   function Set (V : Integer) return Integer is
6     Previous : constant Integer := Value;
7   begin
8     Value := V; -- ERROR
9     return Previous;
10    end Set;

```

(continues on next page)

(continued from previous page)

```

11
12     function Get return Integer is (Value);
13
14 end Bad_Functions;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
bad_functions.ads:2:13: error: function with output global "Value" is not allowed
in SPARK
gnatprove: error during analysis of data and information flow

```

GNATprove detects that function Set has a side effect on global variable Value and issues an error. The correct idiom in SPARK for such a case is to use a procedure with an **out** parameter to return the desired result:

Listing 9: ok_subprograms.ads

```

1 package Ok_Subprograms is
2     procedure Set (V : Integer; Prev : out Integer);
3     function Get return Integer;
4 end Ok_Subprograms;

```

Listing 10: ok_subprograms.adb

```

1 package body Ok_Subprograms is
2
3     Value : Integer := 0;
4
5     procedure Set (V : Integer; Prev : out Integer) is
6     begin
7         Prev := Value;
8         Value := V;
9     end Set;
10
11    function Get return Integer is (Value);
12
13 end Ok_Subprograms;

```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
ok_subprograms.ads:2:32: info: initialization of "Prev" proved

```

With the above restrictions in SPARK, none of the conflicts of side effects that can occur in C can occur in SPARK, and this is guaranteed by flow analysis.

DETECTING UNDEFINED BEHAVIOR

Undefined behavior (and critical unspecified behavior, which we'll treat as undefined behavior) are the plague of C programs. Many rules in MISRA C are designed to avoid undefined behavior, as evidenced by the twenty occurrences of "undefined" in the MISRA C:2012 document.

MISRA C Rule 1.3 is the overarching rule, stating very simply:

"There shall be no occurrence of undefined or critical unspecified behaviour."

The deceptive simplicity of this rule rests on the definition of *undefined or critical unspecified behaviour*. Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable.

It is therefore not surprising that a majority of MISRA C checkers do not make a serious attempt to verify compliance with MISRA C Rule 1.3.

79.1 Preventing Undefined Behavior in SPARK

Since SPARK is a subset of the Ada programming language, SPARK programs may exhibit two types of undefined behaviors that can occur in Ada:

- *bounded error*: when the program enters a state not defined by the language semantics, but the consequences are bounded in various ways. For example, reading uninitialized data can lead to a bounded error, when the value read does not correspond to a valid value for the type of the object. In this specific case, the Ada Reference Manual states that either a predefined exception is raised or execution continues using the invalid representation.
- *erroneous execution*: when the program enters a state not defined by the language semantics, but the consequences are not bounded by the Ada Reference Manual. This is the closest to an undefined behavior in C. For example, concurrently writing through different tasks to the same unprotected variable is a case of erroneous execution.

Many cases of undefined behavior in C would in fact raise exceptions in SPARK. For example, accessing an array beyond its bounds raises the exception `Constraint_Error` while reaching the end of a function without returning a value raises the exception `Program_Error`.

The SPARK Reference Manual defines the SPARK subset through a combination of *legality rules* (checked by the compiler, or the compiler-like phase preceding analysis) and *verification rules* (checked by the formal analysis tool GNATprove). Bounded errors and erroneous execution are prevented by a combination of legality rules and the *flow analysis* part of GNATprove, which in particular detects potential reads of uninitialized data, as described in [Detecting Reads of Uninitialized Data](#) (page 905). The following discussion focuses on how SPARK can verify that no exceptions can be raised.

79.2 Proof of Absence of Run-Time Errors in SPARK

The most common run-time errors are related to misuse of arithmetic (division by zero, overflows, exceeding the range of allowed values), arrays (accessing beyond an array bounds, assigning between arrays of different lengths), and structures (accessing components that are not defined for a given variant).

Arithmetic run-time errors can occur with signed integers, unsigned integers, fixed-point and floating-point (although with IEEE 754 floating-point arithmetic, errors are manifest as special run-time values such as NaN and infinities rather than as exceptions that are raised). These errors can occur when applying arithmetic operations or when converting between numeric types (if the value of the expression being converted is outside the range of the type to which it is being converted).

Operations on enumeration values can also lead to run-time errors; e.g., `T'Pred(T'First)` or `T'Succ(T'Last)` for an enumeration type `T`, or `T'Val(N)` where `N` is an integer value that is outside the range `0 .. T'Pos(T'Last)`.

The Update procedure below contains what appears to be a simple assignment statement, which sets the value of array element `A(I+J)` to `P/Q`.

Listing 1: show_runtime_errors.ads

```

1 package Show_Runtime_Errors is
2
3     type Nat_Array is array (Integer range <>) of Natural;
4     -- The values in subtype Natural are 0 , 1, ... Integer'Last
5
6     procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);
7
8 end Show_Runtime_Errors;
```

Listing 2: show_runtime_errors.adb

```

1 package body Show_Runtime_Errors is
2
3     procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
4     begin
5         A (I + J) := P / Q;
6     end Update;
7
8 end Show_Runtime_Errors;
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_runtime_errors.adb:5:12: medium: overflow check might fail, cannot prove
↳ lower bound for I + J [reason for check: result of addition must fit in a 32-
↳ bits machine integer] [possible fix: add precondition (if J >= 0 then I <=
↳ Integer'Last - J else I >= Integer'First - J) to subprogram at show_runtime_
↳ errors.ads:6]
show_runtime_errors.adb:5:12: medium: array index check might fail [reason for
↳ check: result of addition must be a valid index into the array] [possible fix:
↳ add precondition (if J >= 0 then I <= A'Last - J else I >= A'First - J) to
↳ subprogram at show_runtime_errors.ads:6]
show_runtime_errors.adb:5:22: medium: divide by zero might fail [possible fix: add
↳ precondition (Q /= 0) to subprogram at show_runtime_errors.ads:6]
show_runtime_errors.adb:5:22: medium: overflow check might fail, cannot prove
↳ lower bound for P / Q [reason for check: result of division must fit in a 32-
↳ bits machine integer] [possible fix: add precondition (P / Q in Integer) to
```

(continues on next page)

(continued from previous page)

```

↳subprogram at show_runtime_errors.ads:6]
show_runtime_errors.adb:5:22: medium: range check might fail, cannot prove lower_
↳bound for P / Q [reason for check: result of division must fit in the target_
↳type of the assignment] [possible fix: add precondition (P / Q in Natural) to_
↳subprogram at show_runtime_errors.ads:6]
gnatprove: unproved check messages considered as errors

```

However, for an arbitrary invocation of this procedure, say `Update(A, I, J, P, Q)`, an exception can be raised in a variety of circumstances:

- The computation $I+J$ may overflow, for example if I is `Integer'Last` and J is positive.

```
A (Integer'Last + 1) := P / Q;
```

- The value of $I+J$ may be outside the range of the array A .

```
A (A'Last + 1) := P / Q;
```

- The division P / Q may overflow in the special case where P is `Integer'First` and Q is `-1`, because of the asymmetric range of signed integer types.

```
A (I + J) := Integer'First / -1;
```

- Since the array can only contain non-negative numbers (the element subtype is `Natural`), it is also an error to store a negative value in it.

```
A (I + J) := 1 / -1;
```

- Finally, if Q is 0 then a divide by zero error will occur.

```
A (I + J) := P / 0;
```

For each of these potential run-time errors, the compiler will generate checks in the executable code, raising an exception if any of the checks fail:

```

A (Integer'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
-- raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
-- raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
-- raised CONSTRAINT_ERROR : divide by zero

```

These run-time checks incur an overhead in program size and execution time. Therefore it may be appropriate to remove them if we are confident that they are not needed.

The traditional way to obtain the needed confidence is through testing, but it is well known that this can never be complete, at least for non-trivial programs. Much better is to guarantee the absence of run-time errors through sound static analysis, and that's where SPARK and GNATprove can help.

More precisely, GNATprove logically interprets the meaning of every instruction in the program, taking into account both control flow and data/information dependencies. It uses this analysis to generate a logical formula called a *verification condition* for each possible check.

```

A (Integer'Last + 1) := P / Q;
-- medium: overflow check might fail

A (A'Last + 1) := P / Q;
-- medium: array index check might fail

A (I + J) := Integer'First / (-1);
-- medium: overflow check might fail

A (I + J) := 1 / (-1);
-- medium: range check might fail

A (I + J) := P / 0;
-- medium: divide by zero might fail

```

The verification conditions are then given to an automatic prover. If every verification condition can be proved, then no run-time errors will occur.

GNATprove's analysis is sound — it will detect all possible instances of run-time exceptions being raised — while also having high precision (i.e., not producing a cascade of "false alarms").

The way to program in SPARK so that GNATprove can guarantee the absence of run-time errors entails:

- declaring variables with precise constraints, and in particular to specify precise ranges for scalars; and
- defining preconditions and postconditions on subprograms, to specify respectively the constraints that callers should respect and the guarantees that the subprogram should provide on exit.

For example, here is a revised version of the previous example, which can guarantee through proof that no possible run-time error can be raised:

Listing 3: no_runtime_errors.ads

```

1 package No_Runtime_Errors is
2
3   subtype Index_Range is Integer range 0 .. 100;
4
5   type Nat_Array is array (Index_Range range <>) of Natural;
6
7   procedure Update (A    : in out Nat_Array;
8                     I, J : Index_Range;
9                     P, Q : Positive)
10  with
11    Pre => I + J in A'Range;
12
13 end No_Runtime_Errors;

```

Listing 4: no_runtime_errors.adb

```

1 package body No_Runtime_Errors is
2
3   procedure Update (A    : in out Nat_Array;
4                     I, J : Index_Range;
5                     P, Q : Positive) is
6   begin
7     A (I + J) := P / Q;
8   end Update;
9
10 end No_Runtime_Errors;

```

Prover output

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
no_runtime_errors.adb:7:12: info: index check proved
no_runtime_errors.adb:7:22: info: division check proved
```


DETECTING UNREACHABLE CODE AND DEAD CODE

MISRA C defines *unreachable code* as code that cannot be executed, and it defines *dead code* as code that can be executed but has no effect on the functional behavior of the program. (These definitions differ from traditional terminology, which refers to the first category as "dead code" and the second category as "useless code".) Regardless of the terminology, however, both types are actively harmful, as they might confuse programmers and lead to errors during maintenance.

The "Unused code" section of MISRA C contains seven rules that deal with detecting both unreachable code and dead code. The two most important rules are:

- Rule 2.1: "*A project shall not contain unreachable code*", and
- Rule 2.2: "*There shall not be dead code*".

Other rules in the same section prohibit unused entities of various kinds (type declarations, tag declarations, macro declarations, label declarations, function parameters).

While some simple cases of unreachable code can be detected by static analysis (typically if a condition in an `if` statement can be determined to be always true or false), most cases of unreachable code can only be detected by performing coverage analysis in testing, with the caveat that code reported as not being executed is not necessarily unreachable (it could simply reflect gaps in the test suite). Note that *statement coverage*, rather than the more comprehensive *decision coverage* or *modified condition / decision coverage* (MC/DC) as defined in the DO-178C standard for airborne software, is sufficient to detect potential unreachable statements, corresponding to code that is not covered during the testing campaign.

The presence of dead code is much harder to detect, both statically and dynamically, as it requires creating a complete dependency graph linking statements in the code and their effect on visible behavior of the program.

SPARK can detect some cases of both unreachable and dead code through its precise construction of a dependency graph linking a subprogram's statements to all its inputs and outputs. This analysis might not be able to detect complex cases, but it goes well beyond what other analyses do in general.

Listing 1: much_ado_about_little.ads

```
1 procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean);
```

Listing 2: much_ado_about_little.adb

```
1 procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean) is
2
3     procedure Ok is
4         begin
5             Success := True;
6         end Ok;
```

(continues on next page)

(continued from previous page)

```

8   procedure NOk is
9     begin
10    Success := False;
11   end NOk;

12
13 begin
14   Success := False;

15
16   for K in Y .. Z loop
17     if K < X and not Success then
18       Ok;
19     end if;
20   end loop;

21
22   if X > Y then
23     Ok;
24   else
25     NOk;
26   end if;

27
28   if Z > Y then
29     NOk;
30     return;
31   else
32     Ok;
33     return;
34   end if;

35
36   if Success then
37     Success := not Success;
38   end if;
39 end Much_Ado_About_Little;

```

Build output

```
much_ado_about_little.adb:36:04: warning: unreachable code [enabled by default]
```

Prover output

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
much_ado_about_little.adb:5:15: warning: unused assignment, in call inlined at much_ado_about_little.adb:18
much_ado_about_little.adb:5:15: warning: unused assignment, in call inlined at much_ado_about_little.adb:23
much_ado_about_little.adb:10:15: warning: unused assignment, in call inlined at much_ado_about_little.adb:25
much_ado_about_little.adb:14:12: warning: unused assignment
much_ado_about_little.adb:16:20: warning: statement has no effect
much_ado_about_little.adb:17:07: warning: statement has no effect
much_ado_about_little.adb:22:04: warning: statement has no effect
much_ado_about_little.adb:36:04: warning: unreachable code [enabled by default]
much_ado_about_little.adb:36:04: warning: this statement is never reached
much_ado_about_little.adb:37:15: warning: this statement is never reached
much_ado_about_little.ads:1:34: warning: unused initial value of "X"

```

The only code in the body of `Much_Ado_About_Little` that affects the result of the procedure's execution is the `if Z > Y...` statement, since this statement sets `Success` to either True or False regardless of what the previous statements did. I.e., the statements preceding this `if` are dead code in the MISRA C sense. Since both branches of the `if Z > Y...` statement return from the procedure, the subsequent `if Success...` statement is

unreachable. GNATprove detects and issues warnings about both the dead code and the unreachable code.

CHAPTER
EIGHTYONE

CONCLUSION

The C programming language is "close to the metal" and has emerged as a *lingua franca* for the majority of embedded platforms of all sizes. However, its software engineering deficiencies (such as the absence of data encapsulation) and its many traps and pitfalls present major obstacles to those developing critical applications. To some extent, it is possible to put the blame for programming errors on programmers themselves, as Linus Torvalds admonished:

"Learn C, instead of just stringing random characters together until it compiles (with warnings)."

But programmers are human, and even the best would be hard pressed to be 100% correct about the myriad of semantic details such as those discussed in this document. Programming language abstractions have been invented precisely to help developers focus on the "big picture" (thinking in terms of problem-oriented concepts) rather than low-level machine-oriented details, but C lacks these abstractions. As Kees Cook from the Kernel Self Protection Project puts it (during the Linux Security Summit North America 2018):

"Talking about C as a language, and how it's really just a fancy assembler"

Even experts sometimes have problems with the C programming language rules, as illustrated by Microsoft expert David LeBlanc (see [Enforcing Strong Typing for Scalars](#) (page 896)) or the MISRA C Committee itself (see the [Preface](#) (page 869)).

The rules in MISRA C represent an impressive collective effort to improve the reliability of C code in critical applications, with a focus on avoiding error-prone features rather than enforcing a particular programming style. The Rationale provided with each rule is a clear and unobjectionable justification of the rule's benefit.

At a fundamental level, however, MISRA C is still built on a base language that was not really designed with the goal of supporting large high-assurance applications. As shown in this document, there are limits to what static analysis can enforce with respect to the MISRA C rules. It's hard to retrofit reliability, safety and security into a language that did not have these as goals from the start.

The SPARK language took a different approach, starting from a base language (Ada) that was designed from the outset to support solid software engineering, and eliminating features that were implementation dependent or otherwise hard to formally analyze. In this document we have shown how the SPARK programming language and its associated formal verification tools can contribute usefully to the goal of producing error-free software, going beyond the guarantees that can be achieved in MISRA C.

REFERENCES

82.1 About MISRA C

The official website of the MISRA association <https://www.misra.org.uk/> has many freely available resources about MISRA C, some of which can be downloaded after registering on the MISRA Bulletin Board at <https://www.misra.org.uk/forum/> (such as the examples from the MISRA C:2012 standard, which includes a one-line description of each guideline).

The following documents are freely available:

- *MISRA Compliance 2016: Achieving compliance with MISRA coding guidelines*, 2016, which explains the rationale and process for compliance, including a thorough discussions of acceptable deviations
- *MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012*, 2016, which contains 14 additional guidelines focusing on security. This is a minor addition to MISRA C.

The main MISRA C:2012 document can be purchased from the MISRA webstore.

PRQA is the company that first developed MISRA C, and they have been heavily involved in every version since then. Their webpage <http://www.prqa.com/coding-standards/misra/> contains many resources about MISRA C: product datasheets, white papers, webinars, professional courses.

The PRQA Resources Library at http://info.prqa.com/resources-library?filter=white_paper has some freely available white papers on MISRA C and the use of static analyzers:

- An introduction to MISRA C:2012 at <http://info.prqa.com/MISRA%20C-2012-whitepaper-evaluation-lp>
- *The Myth of Perfect MISRA Compliance* at <http://info.prqa.com/myth-of-perfect-MISRA%20Compliance-evaluation-lp>, providing background information on the use and limitations of static analyzers for checking MISRA C compliance

In 2013 ISO standardized a set of 45 rules focused on security, available in the *C Secure Coding Rules*. A draft is freely available at <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1624.pdf>

In 2018 MISRA published *MISRA C:2012 - Addendum 2: Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure"*, mapping ISO rules to MISRA C:2012 guidelines. This document is freely available from <https://www.misra.org.uk/>.

82.2 About SPARK

The e-learning website <https://learn.adacore.com/> contains a freely available interactive course on SPARK.

The SPARK User's Guide is available at <http://docs.adacore.com/spark2014-docs/html/ug/>.

The SPARK Reference Manual is available at <http://docs.adacore.com/spark2014-docs/html/lrm/>.

A student-oriented textbook on SPARK is *Building High Integrity Applications with SPARK* by John McCormick and Peter Chapin, published by Cambridge University Press. It covers the latest version of the language, SPARK 2014.

A historical account of the evolution of SPARK technology and its use in industry is covered in the article *Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK* by Roderick Chapman and Florian Schanda, at <http://proteancode.com/keynote.pdf>

The website <https://www.adacore.com/sparkpro> is a portal for up-to-date information and resources on SPARK. AdaCore blog's site <https://blog.adacore.com/> contains a number of SPARK-related posts.

The booklet *AdaCore Technologies for Cyber Security* shows how AdaCore's technology can be used to prevent or mitigate the most common security vulnerabilities in software. See <https://www.adacore.com/books/adacore-tech-for-cyber-security/>.

The booklet *AdaCore Technologies for CENELEC EN 50128:2011* shows how AdaCore's technology can be used in conjunction with the CENELEC EN 50128:2011 software standard for railway control and protection systems. It describes in particular where the SPARK technology fits best and how it can be used to meet various requirements of the standard. See: <https://www.adacore.com/books/cenelec-en-50128-2011/>.

The booklet *AdaCore Technologies for DO-178C/ED-12C* similarly shows how AdaCore's technology can be used in conjunction with the DO-178C/ED-12C standard for airborne software, and describes in particular how SPARK can be used in conjunction with the Formal Methods supplement DO-333/ED-216. See <https://www.adacore.com/books/do-178c-tech/>.

82.3 About MISRA C and SPARK

The blog post at <https://blog.adacore.com/MISRA-C-2012-vs-spark-2014-the-subset-matching-game> reviews the 27 undecidable rules in MISRA C:2012 and describes how SPARK addresses them.

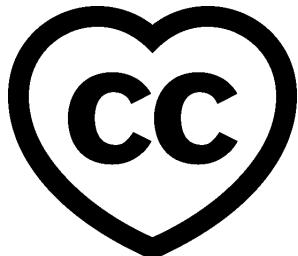
The white paper *A Comparison of SPARK with MISRA C and Frama-C* at <https://www.adacore.com/papers/compare-spark-MISRA-C-frama-c> compares SPARK to MISRA C and to the formal verification tool Frama-C for C programs.

Part VIII

Introduction to GNAT Toolchain

Copyright © 2019 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page](#)¹¹⁹



This course presents an introduction to the GNAT toolchain, which is included in the GNAT Community 2020 edition. The course includes first steps to get started with the toolchain and some details on the project manager (GPRbuild) and the integrated development environment (GNAT Studio).

This document was written by Gustavo A. Hoffmann, with contributions and review from Richard Kenner and Robert Duff.

¹¹⁹ <http://creativecommons.org/licenses/by-sa/4.0>

GNAT COMMUNITY

This chapter presents the steps needed to install the GNAT Community toolchain and how to use basic commands from the toolchain.

83.1 Installation

These are the basics steps to install GNAT Community on all platforms:

- Go to the AdaCore Community page¹²⁰.
- Download the GNAT installer.
- Run the GNAT installer.
 - Leave all options checked on the "Select Components" page.

On Windows platforms, continue with the following steps:

- Add C:\GNAT\2020\bin to your Path environment variable.
 - The environment variables can be found in the System Properties window of the Control Panel.
- You might need to restart your computer for the settings to take effect.

On Linux platforms, perform the following steps:

- Make sure the GNAT installer has execution permissions before running it.
- Select the directory where you want to install the toolchain.
 - For example: /home/me/GNAT/2020
- Add the path to the bin directory (within the toolchain directory) as the first directory in your PATH environment variable.
 - For example: /home/me/GNAT/2020/bin.

¹²⁰ <https://www.adacore.com/community>

83.2 Basic commands

Now that the toolchain is installed, you can start using it. From the command line, you can compile a project using **gprbuild**. For example:

```
gprbuild -P project.gpr
```

You can find the binary built with the command above in the *obj* directory. You can run it in the same way as you would do with any other executable on your platform. For example:

```
obj/main
```

A handy command-line option for **gprbuild** you might want to use is **-p**, which automatically creates directories such as *obj* if they aren't in the directory tree:

```
gprbuild -p -P project.gpr
```

Ada source-code are stored in *.ads* and *.adb* files. To view the content of these files, you can use **GNAT Studio**. To open **GNAT Studio**, double-click on the *.gpr* project file or invoke **GNAT Studio** on the command line:

```
gps -P project.gpr
```

To compile your project using **GNAT Studio**, use the top-level menu to invoke Build → Project → main.adb (or press the keyboard shortcut F4). To run the main program, click on Build → Run → main (or press the keyboard shortcut Shift + F2).

83.3 Compiler warnings

One of the strengths of the GNAT compiler is its ability to generate many useful warnings. Some are displayed by default but others need to be explicitly enabled. In this section, we discuss some of these warnings, their purpose, and how you activate them.

83.3.1 -gnatwa switch and warning suppression

Section author: Robert Duff

We first need to understand the difference between a *warning* and an *error*. Errors are violations of the Ada language rules as specified in the Ada Reference Manual; warnings don't indicate violations of those rules, but instead flag constructs in a program that seem suspicious to the compiler. Warnings are GNAT-specific, so other Ada compilers might not warn about the same things GNAT does or might warn about them in a different way. Warnings are typically conservative; meaning that some warnings are false alarms. The programmer needs to study the code to determine if each warning is describing a real problem.

Some warnings are produced by default while others are produced only if a switch enables them. Use the **-gnatwa** switch to turn on (almost) all warnings.

Warnings are useless if you don't do anything about them. If you give your team member some code that causes warnings, how are they supposed to know whether they represent real problems? If you don't address each warning, people will soon start ignoring warnings and there'll be lots of things that generates warnings scattered all over your code. To avoid this, you may want to use the **-gnatwae** switch to both turn on (almost) all warnings and to treat warnings as errors. This forces you to get a clean (no warnings or errors) compilation.

However, as we said, some warnings are false alarms. Use `pragma Warnings (Off)` to suppress those warnings. It's best to be as specific as possible and narrow down to a single line of code and a single warning. Then use a comment to explain why the warning is a false alarm if it's not obvious.

Let's look at the following example:

```
with Ada.Text_IO; use Ada.Text_IO;

package body Warnings_Example is

    procedure Mumble (X : Integer) is
    begin
        Put_Line ("Mumble processing...");
    end Mumble;

end Warnings_Example;
```

We compile the above code with `-gnatwae`:

```
gnat compile -gnatwae ./src/warnings_example.adb
```

This causes GNAT to complain:

```
warnings_example.adb:5:22: warning: formal parameter "X" is not referenced
```

But the following compiles cleanly:

```
with Ada.Text_IO; use Ada.Text_IO;

package body Warnings_Example is

    pragma Warnings (Off, "formal parameter ""X"" is not referenced");
    procedure Mumble (X : Integer) is
    pragma Warnings (On, "formal parameter ""X"" is not referenced");

    -- X is ignored here, because blah blah blah...
    begin
        Put_Line ("Mumble processing...");
    end Mumble;

end Warnings_Example;
```

Here we've suppressed a specific warning message on a specific line.

If you get many warnings of a specific type and it's not feasible to fix all of them, you can suppress that type of message so the good warnings won't get buried beneath a pile of bogus ones. For example, you can use the `-gnatwaeF` switch to silence the warning on the first version of `Mumble` above: the `F` suppresses warnings on unreferenced formal parameters. It would be a good idea to use it if you have many of those.

As discussed above, `-gnatwa` activates almost all warnings, but not all. Refer to the [section on warnings¹²¹](#) of the GNAT User's Guide to get a list of the remaining warnings you could enable in your project. One is `-gnatw.o`, which displays warnings when the compiler detects modified but unreferenced `out` parameters. Consider the following example:

```
package Warnings_Example is

    procedure Process (X : in out Integer;
                      B :      out Boolean);
```

(continues on next page)

¹²¹ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#warning-message-control

(continued from previous page)

```
end Warnings_Example;

package body Warnings_Example is

    procedure Process (X : in out Integer;
                       B :      out Boolean) is
begin
    if X = Integer'First or else X = Integer'Last then
        B := False;
    else
        X := X + 1;
        B := True;
    end if;
end Process;

end Warnings_Example;
```

```
with Ada.Text_IO; use Ada.Text_IO;

with Warnings_Example; use Warnings_Example;

procedure Main  is
    X : Integer := 0;
    Success : Boolean;
begin
    Process (X, Success);
    Put_Line (Integer'Image (X));
end Main;
```

If we build the main application using the -gnatw.o switch, the compiler warns us that we didn't reference the Success variable, which was modified in the call to Process:

```
main.adb:8:16: warning: "Success" modified by call, but value might not be ↴ referenced
```

In this case, this actually points us to a bug in our program, since X only contains a valid value if Success is **True**. The corrected code for Main is:

```
-- ...
begin
    Process (X, Success);

    if Success then
        Put_Line (Integer'Image (X));
    else
        Put_Line ("Couldn't process variable X.");
    end if;
end Main;
```

We suggest turning on as many warnings as makes sense for your project. Then, when you see a warning message, look at the code and decide if it's real. If it is, fix the code. If it's a false alarm, suppress the warning. In either case, we strongly recommend you make the warning disappear before you check your code into your configuration management system.

83.3.2 Style checking

GNAT provides many options to configure style checking of your code. The main compiler switch for this is `-gnatyy`, which sets almost all standard style check options. As indicated by the [section on style checking¹²²](#) of the GNAT User's Guide, using this switch "is equivalent to `-gnaty3aAbcefhiklmnprst`, that is all checking options enabled with the exception of `-gnatyB`, `-gnatyd`, `-gnatyI`, `-gnatyLnnn`, `-gnatyo`, `-gnaty0`, `-gnatyS`, `-gnatyu`, and `-gnatyx`."

You may find that selecting the appropriate coding style is useful to detect issues at early stages. For example, the `-gnaty0` switch checks that overriding subprograms are explicitly marked as such. Using this switch can avoid surprises when you didn't intentionally want to override an operation for some data type. We recommend studying the list of coding style switches and selecting the ones that seem relevant for your project. When in doubt, you can start by using all of them — using `-gnatyy` and `-gnatyBdIL4o0Sux`, for example — and deactivating the ones that cause too much *noise* during compilation.

¹²² https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/building_executable_programs_with_gnat.html#style-checking

GPRBUILD

This chapter presents a brief overview of **GPRbuild**, the project manager of the GNAT toolchain. It can be used to manage complex builds. In terms of functionality, it's similar to **make** and **cmake**, just to name two examples.

For a detailed presentation of the tool, please refer to the [GPRbuild User's Guide](#)¹²³.

84.1 Basic commands

As mentioned in the previous chapter, you can build a project using **gprbuild** from the command line:

```
gprbuild -P project.gpr
```

In order to clean the project, you can use **gprclean**:

```
gprclean -P project.gpr
```

84.2 Project files

You can create project files using **GNAT Studio**, which presents many options on its graphical interface. However, you can also edit project files manually as a normal text file in an editor, since its syntax is human readable. In fact, project files use a syntax similar to the one from the Ada language. Let's look at the basic structure of project files and how to customize them.

84.2.1 Basic structure

The main element of a project file is a project declaration, which contains definitions for the current project. A project file may also include other project files in order to compose a complex build. One of the simplest form of a project file is the following:

```
project Default is
  for Main use ("main");
  for Source_Dirs use ("src");
end Default;
```

¹²³ https://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html

In this example, we declare a project named Default. The for Main use expression indicates that the main.adb file is used as the entry point (main source-code file) of the project. The main file doesn't necessarily need to be called main.adb; we could use any source-code implementing a main application, or even have a list of multiple main files. The for Source_Dirs use expression indicates that the src directory contains the source-file for the application (including the main file).

84.2.2 Customization

GPRbuild support scenario variables, which allow you to control the way binaries are built. For example, you may want to distinguish between debug and optimized versions of your binary. In principle, you could pass command-line options to **gprbuild** that turn debugging on and off, for example. However, defining this information in the project file is usually easier to handle and to maintain. Let's define a scenario variable called ver in our project:

```
project Default is
    Ver := external ("ver", "debug");
    for Main use ("main");
    for Source_Dirs use ("src");
end Default;
```

In this example, we're specifying that the scenario variable Ver is initialized with the external variable ver. Its default value is set to debug.

We can now set this variable in the call to **gprbuild**:

```
gprbuild -P project.gpr -Xver=debug
```

Alternatively, we can simply specify an environment variable. For example, on Unix systems, we can say:

```
export ver=debug
# Value from environment variable "ver" used in the following call:
gprbuild -P project.gpr
```

In the project file, we can use the scenario variable to customize the build:

```
project Default is
    Ver := external ("ver", "debug");

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

    -- Using "ver" variable for obj directory
    for Object_Dir use "obj/" & Ver;

    package Compiler is
        case Ver is
            when "debug" =>
                for Switches ("Ada") use ("-g");
            when "opt" =>
                for Switches ("Ada") use ("-O2");
            when others =>
                null;
        end case;
```

(continues on next page)

(continued from previous page)

```

end Compiler;

end Default;

```

We're now using `Ver` in the `for Object_Dir` clause to specify a subdirectory of the `obj` directory that contains the object files. Also, we're using `Ver` to select compiler options in the `Compiler` package declaration.

We could also specify all available options in the project file by creating a typed variable. For example:

```

project Default is

    type Ver_Option is ("debug", "opt");
    Ver : Ver_Option := external ("ver", "debug");

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

    -- Using "ver" variable for obj directory
    for Object_Dir use "obj/" & Ver;

    package Compiler is
        case Ver is
            when "debug" =>
                for Switches ("Ada") use ("-g");
            when "opt" =>
                for Switches ("Ada") use ("-O2");
            when others =>
                null;
        end case;
    end Compiler;

end Default;

```

The advantage of this approach is that `gprbuild` can now check whether the value that you provide for the `ver` variable is available on the list of possible values and give you an error if you're entering a wrong value.

84.3 Project dependencies

GPRbuild supports project dependencies. This allows you to reuse information from existing projects. Specifically, the keyword `with` allows you to include another project within the current project.

84.3.1 Simple dependency

Let's look at a very simple example. We have a package called `Test_Pkg` associated with the project file `test_pkg.gpr`, which contains:

```

project Test_Pkg is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Test_Pkg;

```

This is the code for the `Test_Pkg` package:

```
package Test_Pkg is

    type T is record
        X : Integer;
        Y : Integer;
    end record;

    function Init return T;

end Test_Pkg;
```

```
package body Test_Pkg is

    function Init return T is
    begin
        return V : T do
            V.X := 0;
            V.Y := 0;
        end return;
    end Init;

end Test_Pkg;
```

For this example, we use a directory `test_pkg` containing the project file and a subdirectory `test_pkg/src` containing the source files. The directory structure looks like this:

```
|- test_pkg
|   | test_pkg.gpr
|   |- src
|       | test_pkg.adb
|       | test_pkg.ads
```

Suppose we want to use the `Test_Pkg` package in a new application. Instead of directly including the source files of `Test_Pkg` in the project file of our application (either directly or indirectly), we can instead reference the existing project file for the package by using `with "test_pkg.gpr"`. This is the resulting project file:

```
with "../test_pkg/test_pkg.gpr";

project Default is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
    for Main use ("main.adb");
end Default;
```

And this is the code for the main application:

```
with Test_Pkg; use Test_Pkg;

procedure Main is
    A : T;
begin
    A := Init;
end Main;
```

When we build the main project file (`default.gpr`), we're automatically building all dependent projects. More specifically, the project file for the main application automatically includes the information from the dependent projects such as `test_pkg.gpr`. Using a `with` in the main project file is all we have to do for that to happen.

84.3.2 Dependencies to dynamic libraries

We can structure project files to make use of dynamic (shared) libraries using a very similar approach. It's straightforward to convert the project above so that `Test_Pkg` is now compiled into a dynamic library and linked to our main application. All we need to do is to make a few additions to the project file for the `Test_Pkg` package:

```
library project Test_Pkg is
  for Source_Dirs use ("src");
  for Object_Dir use "obj";
  for Library_Name use "test_pkg";
  for Library_Dir use "lib";
  for Library_Kind use "Dynamic";
end Test_Pkg;
```

This is what we had to do:

- We changed the project to `library` project.
- We added the specification for `Library_Name`, `Library_Dir` and `Library_Kind`.

We don't need to change the project file for the main application because **GPRbuild** automatically detects the dependency information (e.g., the path to the dynamic library) from the project file for the `Test_Pkg` package. With these small changes, we're able to compile the `Test_Pkg` package to a dynamic library and link it with our main application.

84.4 Configuration pragma files

Configuration pragma files contain a set of pragmas that modify the compilation of source files according to external requirements. For example, you may use pragmas to either relax or strengthen requirements depending on your environment.

In **GPRbuild**, we can use `Local_Configuration_Pragmas` (in the `Compiler` package) to indicate the configuration pragmas file we want **GPRbuild** to use with the source files in our project.

The file `gnat.adc` shown here is an example of a configuration pragma file:

```
pragma Suppress (Overflow_Check);
```

We can use this in our project by declaring a `Compiler` package. Here's the complete project file:

```
project Default is

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

  package Compiler is
    for Local_Configuration_Pragmas use "gnat.adc";
  end Compiler;

end Default;
```

Each pragma contained in `gnat.adc` is used in the compilation of each file, as if that pragma was placed at the beginning of each file.

84.5 Configuration packages

You can control the compilation of your source code by creating variants for various cases and selecting the appropriate variant in the compilation package in the project file. One example where this is useful is conditional compilation using Boolean constants, shown in the code below:

```
with Ada.Text_IO; use Ada.Text_IO;

with Config;

procedure Main is
begin

  if Config.Debug then
    Put_Line ("Debug version");
  else
    Put_Line ("Release version");
  end if;
end Main;
```

In this example, we declared the Boolean constant in the Config package. By having multiple versions of that package, we can create different behavior for each usage. For this simple example, there are only two possible cases: either Debug is **True** or **False**. However, we can apply this strategy to create more complex cases.

In our next example, we store the packages in the subdirectories debug and release of the source code directory. Here's the content of the src/debug/config.ads file:

```
package Config is

  Debug : constant Boolean := True;

end Config;
```

Here's the src/release/config.ads file:

```
package Config is

  Debug : constant Boolean := False;

end Config;
```

In this case, **GPRbuild** selects the appropriate directory to look for the config.ads file according to information we provide for the compilation process. We do this by using a scenario type called **Mode_Type** in our project file:

```
gprbuild -P default.gpr -Xmode=release
```

```
project Default is

  type Mode_Type is ("debug", "release");

  Mode : Mode_Type := external ("mode", "debug");

  for Source_Dirs use ("src", "src/" & Mode);
  for Object_Dir use "obj";
  for Main use ("main.adb");

end Default;
```

We declare the scenario variable Mode and use it in the Source_Dirs declaration to add the desired path to the subdirectory containing the config.ads file. The expression "src/" & Mode concatenates the user-specified mode to select the appropriate subdirectory. For more complex cases, we could use either a tree of subdirectories or multiple scenario variables for each aspect that we need to configure.

GNAT STUDIO

This chapter presents an introduction to the GNAT Studio, which provides an IDE to develop applications in Ada. For a detailed overview, please refer to the [GNAT Studio tutorial¹²⁴](#). Also, you can refer to the [GNAT Studio product page¹²⁵](#) for some introductory videos.

In this chapter, all indications using "→" refer to options from the GNAT Studio menu that you can click in order to execute commands.

85.1 Start-up

The first step is to start-up the GNAT Studio. The actual step depends on your platform.

85.1.1 Windows

- You may find an icon (shortcut to **GNAT Studio**) on your desktop.
- Otherwise, start **GNAT Studio** by typing gnatstudio on the command prompt.

85.1.2 Linux

- Start **GNAT Studio** by typing gnatstudio on a shell.

85.2 Creating projects

After starting-up **GNAT Studio**, you can create a project. These are the steps:

- Click on Create new project in the welcome window
 - Alternatively, if the *wizard* (which let's you customize new projects) isn't already opened, click on File → New Project... to open it.
 - After clicking on Create new project, you should see a window with this title: Create Project from Template.
- Select one of the options from the list and click on Next.
 - The simplest one is Basic > Simple Ada Project, which creates a project containing a main application.
- Select the project location and basic settings, and click on Apply.

¹²⁴ https://docs.adacore.com/live/wave/gps/html/gps_tutorial/index.html

¹²⁵ <https://www.adacore.com/gnatpro/toolsuite/gps>

- If you selected "Simple Ada Project" in the previous step, you may now select the name of the project and of the main file.
- Note that you can select any name for the main file.

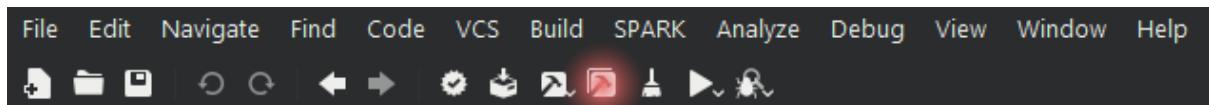
You should now have a working project file.

85.3 Building

As soon as you've created a project file, you can use it to build an application. These are the required steps:

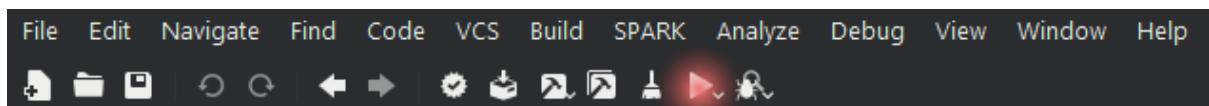
- Click on Build → Project → Build All

- You can also click on this icon:



- Alternatively, you can click on Build → Project → Build & Run → <name of your main application>

- You can also click on this icon:



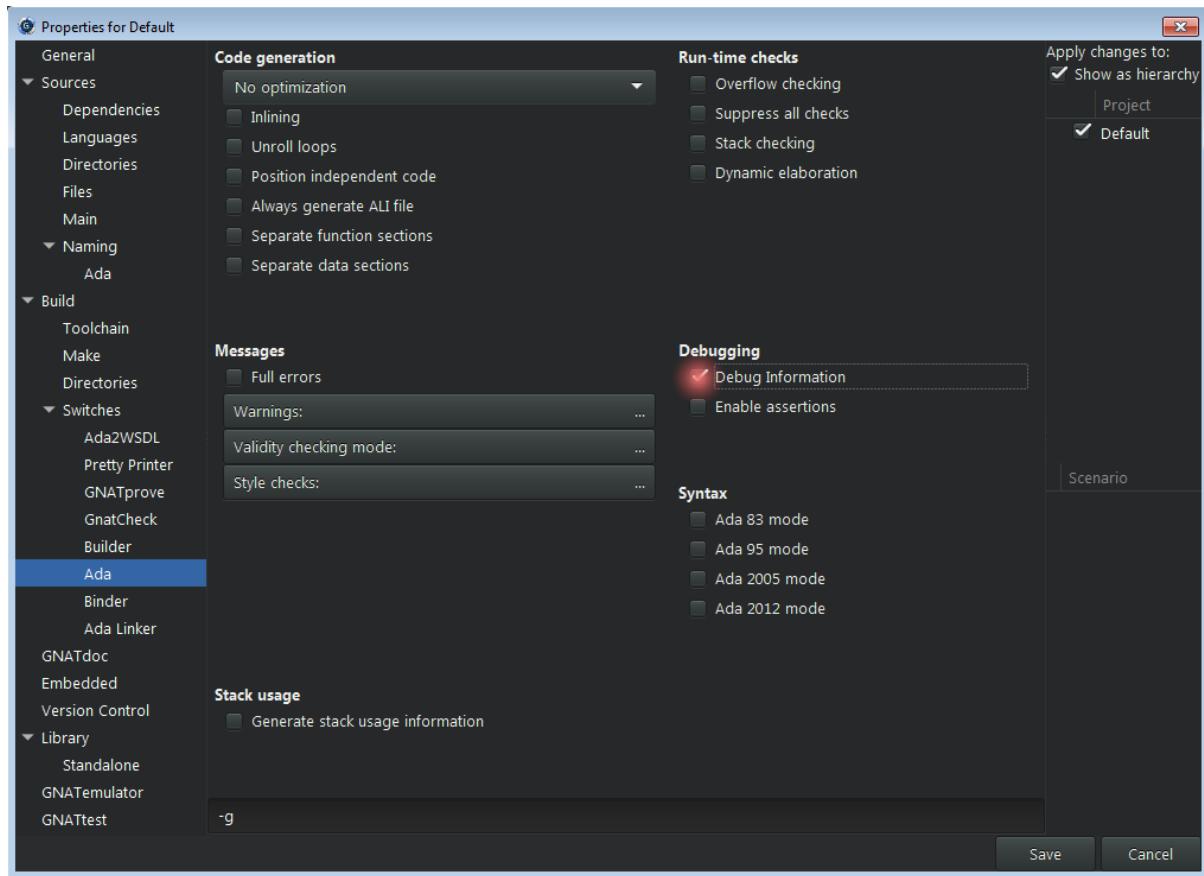
- You can also use the keyboard for building and running the main application:
 - Press F4 to open a window that allows you to build the main application and click on Execute.
 - Then, press Shift + F2 to open a window that allows you to run the application, and click on Execute.

85.4 Debugging

85.4.1 Debug information

Before you can debug a project, you need to make sure that debugging symbols have been included in the binary build. You can do this by manually adding a debug version into your project, as described in the previous chapter (see [GPRbuild](#) (page 943)).

Alternatively, you can change the project properties directly in **GNAT Studio**. In order to do that, click on Edit → Project Properties..., which opens the following window:



Click on Build → Switches → Ada on this window, and make sure that the Debug Information option is selected.

85.4.2 Improving main application

If you selected "Simple Ada Project" while creating your project in the beginning, you probably still have a very simple main application that doesn't do anything useful. Therefore, in order to make the debugging activity more interesting, please enter some statements to your application. For example:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
begin
    Put_Line ("Hello World!");
    Put_Line ("Hello again!");
end Main;
```

85.4.3 Debugging the application

You can now build and debug the application by clicking on Build → Project → Build & Debug → <name of your main application>.

You can then click on Debug → Run... to open a window that allows you to start the application. Alternatively, you can press Shift + F9. As soon as the application has started, you can press F5 to step through the application or press F6 to execute until the next line. Both commands are available in the menu by clicking on Debug → Step or Debug → Next.

When you've finished debugging your application, you need to terminate the debugger. To do this, you can click on Debug → Terminate.

85.5 Formal verification

In order to see how SPARK can detect issues, let's creating a simple application that accumulates values in a variable A:

```
procedure Main
  with SPARK_Mode is

  procedure Acc (A : in out Natural;
                V :           Natural) is
    begin
      A := A + V;
    end Acc;

  A : Natural := 0;
begin
  Acc (A, Natural'Last);
  Acc (A, 1);
end Main;
```

You can now click on SPARK → Prove All, which opens a window with various options. For example, on this window, you can select the proof level — varying between 0 and 4 — on the Proof level list. Next, click on Execute. After the prover has completed its analysis, you'll see a list of issues found in the source code of your application.

For the example above, the prover complains about an overflow check that might fail. This is due to the fact that, in the Acc procedure, we're not dealing with the possibility that the result of the addition might be out of range. In order to fix this, we could define a new saturating addition Sat_Add that makes use of a custom type T with an extended range. For example:

```
procedure Main
  with SPARK_Mode is

  function Sat_Add (A : Natural;
                    V : Natural) return Natural
  is
    type T is range Natural'First .. Natural'Last * 2;

    A2      : T      := T (A);
    V2      : constant T := T (V);
    A_Last : constant T := T (Natural'Last);
begin
  A2 := A2 + V2;
  -- Saturate result if needed
```

(continues on next page)

(continued from previous page)

```
if A2 > A_Last then
    A2 := A_Last;
end if;

return Natural (A2);
end Sat_Add;

procedure Acc (A : in out Natural;
               V :          Natural) is
begin
    A := Sat_Add (A, V);
end Acc;

A : Natural := 0;
begin
    Acc (A, Natural'Last);
    Acc (A, 1);
end Main;
```

Now, when running the prover again with the modified code, no issues are found.

GNAT TOOLS

In chapter we present a brief overview of some of the tools included in the GNAT Community toolchain.

For further details on how to use these tools, please refer to the [GNAT User's Guide](#)¹²⁶.

86.1 gnatchop

gnatchop renames files so they match the file structure and naming convention expected by the rest of the GNAT toolchain. The GNAT compiler expects specifications to be stored in .ads files and bodies (implementations) to be stored in .adb files. It also expects file names to correspond to the content of each file. For example, it expects the specification of a package Pkg.Child to be stored in a file named pkg-child.ads.

However, we may not want to use that convention for our project. For example, we may have multiple Ada packages contained in a single file. Consider a file example.ada containing the following:

```
with Ada.Text_IO; use Ada.Text_IO;

package P is
    procedure Test;
end P;

package body P is
    procedure Test is
    begin
        Put_Line("Test passed.");
    end Test;
end P;

with P; use P;

procedure P_Main is
begin
    P.Test;
end P_Main;
```

To compile this code, we first pass the file containing our source code to **gnatchop** before we call **gprbuild**:

```
gnatchop example.ada
gprbuild p_main
```

This generates source files for our project, extracted from example_ada, that conform to the default naming convention and then builds the executable binary p_main from those

¹²⁶ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn.html

files. In this example **gnatchop** created the files p.ads, p.adb, and p_main.adb using the package names in example.ada.

When we use this mechanism, any warnings or errors the compiler displays refers to the files generated by **gnatchop**. We can, however, instruct **gnatchop** to instrument the generated files so the compiler refers to the original file (example.ada in our case) when displaying messages. We do this by using the -r switch:

```
gnatchop -r example.ada  
gprbuild p_main
```

If, for example, we had an unused variable in example.ada, the compiler warning would now refer to the line in the original file, not in one of the generated ones.

For documentation of other switches available for **gnatchop**, please refer to the [gnatchop chapter¹²⁷](#) of the GNAT User's Guide.

86.2 gnatprep

We may want to use conditional compilation in some situations. For example, we might need a customized implementation of a package for a specific platform or need to select a specific version of an algorithm depending on the requirements of the target environment. A traditional way to do this uses a source-code preprocessor. However, in many cases where conditional compilation is needed, we can instead use the syntax of the Ada language or the functionality provided by **GPRbuild** to avoid using a preprocessor in those cases. The [conditional compilation section¹²⁸](#) of the GNAT User's Guide discusses how to do this in detail.

Nevertheless, using a preprocessor is often the most straightforward option in complex cases. When we encounter such a case, we can use **gnatprep**, which provides a syntax that reminds us of the C and C++ preprocessor. However, unlike in C and C++, this syntax is not part of the Ada standard and can only be used with **gnatprep**. Also, you'll notice some differences in the syntax from that preprocessor, such as shown in the example below:

```
#if VERSION'Defined and then (VERSION >= 4) then  
    -- Implementation for version 4.0 and above...  
#else  
    -- Standard implementation for older versions...  
#end if;
```

Of course, in this simple case, we could have used the Ada language directly and avoided the preprocessor entirely:

```
package Config is  
    Version : constant Integer := 4;  
end Config;  
  
with Config;  
procedure Do_Something is  
begin  
    if Config.Version >= 4 then  
        null;  
        -- Implementation for version 4.0 and above...  
    else  
        null;
```

(continues on next page)

¹²⁷ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#renaming-files-with-gnatchop

¹²⁸ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#conditional-compilation

(continued from previous page)

```
-- Standard implementation for older versions...
end if;
end Do_Something;
```

But for the sake of illustrating the use of **gnatprep**, let's use that tool in this simple case. This is the complete procedure, which we place in file do_something.org.adb:

```
procedure Do_Something is
begin
  #if VERSION'Defined and then (VERSION >= 4) then
    -- Implementation for version 4.0 and above...
    null;
  #else
    -- Standard implementation for older versions...
    null;
  #end if;
end Do_Something;
```

To preprocess this file and build the application, we call **gnatprep** followed by **GPRbuild**:

```
gnatprep do_something.org.adb do_something.adb
gprbuild do_something
```

If we look at the resulting file after preprocessing, we see that the **#else** implementation was selected by **gnatprep**. To cause it to select the newer "version" of the code, we include the symbol and its value in our call to **gnatprep**, just like we'd do for C/C++:

```
gnatprep -DVERSION=5 do_something.org.adb do_something.adb
```

However, a cleaner approach is to create a symbol definition file containing all symbols we use in our implementation. Let's create the file and name it prep.def:

```
VERSION := 5
```

Now we just need to pass it to **gnatprep**:

```
gnatprep do_something.org.adb do_something.adb prep.def
gprbuild do_something
```

When we use **gnatprep** in that way, the line numbers of the output file differ from those of the input file. To preserve line numbers, we can use one of these command-line switches:

- -b: replace stripped-out code by blank lines
- -c: comment-out the stripped-out code

For example:

```
gnatprep -b do_something.org.adb do_something.adb prep.def
gnatprep -c do_something.org.adb do_something.adb prep.def
```

When we use one of these options, **gnatprep** ensures that the output file do_something.adb has the same line numbering as the original file (do_something.org.adb).

The **gnatprep** chapter¹²⁹ of the GNAT User's Guide contains further details about this tool, such as how to integrate **gnatprep** with project files for **GPRbuild** and how to replace symbols without using preprocessing directives (using the \$symbol syntax).

¹²⁹ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/the_gnat_compilation_model.html#preprocessing-with-gnatprep

86.3 gnatmem

Memory allocation errors involving mismatches between allocations and deallocations are a common source of memory leaks. To test an application for memory allocation issues, we can use **gnatmem**. This tool monitors all memory allocations in our application. We use this tool by linking our application to a special version of the memory allocation library (`libgmem.a`).

Let's consider this simple example:

```
procedure Simple_Mem is
  I_Ptr : access Integer := new Integer;
begin
  null;
end Simple_Mem;
```

To generate a memory report for this code, we need to:

- Build the application, linking it to `libgmem.a`;
- Run the application, which generates an output file (`gmem.out`);
- Run **gnatmem** to generate a report from `gmem.out`.

For our example above, we do the following:

```
# Build application using gmem
gnatmake -g simple_mem.adb -largs -lgmem

# Run the application and generate gmem.out
./simple_mem

# Call gnatmem to display the memory report based on gmem.out
gnatmem simple_mem
```

For this example, **gnatmem** produces the following output:

```
Global information
-----
Total number of allocations      :  1
Total number of deallocations    :  0
Final Water Mark (non freed mem) : 4 Bytes
High Water Mark                 : 4 Bytes

Allocation Root # 1
-----
Number of non freed allocations   :  1
Final Water Mark (non freed mem) : 4 Bytes
High Water Mark                 : 4 Bytes
Backtrace                         :
  simple_mem.adb:2 simple_mem
```

This shows all the memory we allocated and tells us that we didn't deallocate any of it.

Please refer to the [chapter on gnatmem¹³⁰](#) of the GNAT User's Guide for a more detailed discussion of **gnatmem**.

¹³⁰ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_and_program_execution.html#the-gnatmem-tool

86.4 gnatmetric

We can use the GNAT metric tool (**gnatmetric**) to compute various programming metrics, either for individual files or for our complete project.

For example, we can compute the metrics of the body of package P above by running **gnatmetric** as follows:

```
gnatmetric p.adb
```

This produces the following output:

```
Line metrics summed over 1 units
  all lines      : 13
  code lines     : 11
  comment lines   : 0
  end-of-line comments : 0
  comment percentage : 0.00
  blank lines     : 2

Average lines in body: 4.00

Element metrics summed over 1 units
  all statements   : 2
  all declarations : 3
  logical SLOC      : 5

2 subprogram bodies in 1 units

Average cyclomatic complexity: 1.00
```

Please refer to the [section on gnatmetric¹³¹](#) of the GNAT User's Guide for the many switches available for **gnatmetric**, including the ability to generate reports in XML format.

86.5 gnatdoc

Use **GNATdoc** to generate HTML documentation for your project. It scans the source files in the project and extracts information from package, subprogram, and type declarations.

The simplest way to use it is to provide the name of the project or to invoke **GNATdoc** from a directory containing a project file:

```
gnatdoc -P some_directory/default.gpr

# Alternatively, when the :file:`default.gpr` file is in the same directory

gnatdoc
```

Just using this command is sufficient if your goal is to generate a list of the packages and a list of subprograms in each. However, to create more meaningful documentation, you can annotate your source code to add a description of each subprogram, parameter, and field. For example:

```
package P is
-- Collection of auxiliary subprograms
```

(continues on next page)

¹³¹ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-metrics-tool-gnatmetric

(continued from previous page)

```
function Add_One
  (V : Integer
   -- Coefficient to be incremented
   ) return Integer;
-- @return Coefficient incremented by one

end P;
```

```
package body P is

  function Add_One (V : Integer) return Integer is
begin
  return V + 1;
end Add_One;

end P;
```

```
with P; use P;

procedure Main is

  I : Integer;

begin
  I := Add_One (0);
end Main;
```

When we run this example, **GNATdoc** will extract the documentation from the specification of package P and add the description of each element, which we provided as a comment in the line below the actual declaration. It will also extract the package description, which we wrote as a comment in the line right after **package P is**. Finally, it will extract the documentation of function Add_One (both the description of the V parameter and the return value).

In addition to the approach we've just seen, **GNATdoc** also supports the tagged format that's commonly found in tools such as Javadoc and uses the @ syntax. We could rewrite the documentation for package P as follows:

```
package P is
-- @summary Collection of auxiliary subprograms

  function Add_One
    (V : Integer
     ) return Integer;
-- @param V Coefficient to be incremented
-- @return Coefficient incremented by one

end P;
```

You can control what parts of the source-code **GNATdoc** parses to extract the documentation. For example, you can specify the -b switch to request that the package body be parsed for additional documentation and you can use the -p switch to request **GNATdoc** to parse the private part of package specifications. For a complete list of switches, please refer to the **GNATdoc User's Guide**¹³².

¹³² http://docs.adacore.com/gnatdoc-docs/users_guide/_build/html/index.html

86.6 gnatpp

The term 'pretty-printing' refers to the process of formatting source code according to a pre-defined convention. **gnatpp** is used for the pretty-printing of Ada source-code files.

Let's look at this example, which contains very messy formatting:

```
PrOcEDuRE Main
IS
  FUNCTioN
    Init_2
    RETurn
  inteGER      iS
                (2);
    I : INTeger;

  BeGiN
    I           :=      Init_2;
  ENd;
```

We can request **gnatpp** to clean up this file by using the command:

```
gnatpp main.adb
```

gnatpp reformats the file in place. After this command, `main.adb` looks like this:

```
procedure Main is
  function Init_2 return Integer iS (2);
  I : Integer;
begin
  I := Init_2;
end Main;
```

We can also process all source code files from a project at once by specifying a project file. For example:

```
gnatpp -P default.gpr
```

gnatpp has an extensive list of options, which allow for specifying the formatting of many aspects of the source and implementing many coding styles. These are extensively discussed in the [section on gnatpp¹³³](#) of the GNAT User's Guide.

¹³³ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-pretty-printer-gnatpp

86.7 gnatstub

Suppose you've created a complex specification of an Ada package. You can create the corresponding package body by copying and adapting the content of the package specification. But you can also have **gnatstub** do much of that job for you. For example, let's consider the following package specification:

```
package Aux is
    function Add_One (V : Integer) return Integer;
    procedure Reset (V : in out Integer);
end Aux;
```

We call **gnatstub**, passing the file containing the package specification:

```
gnatstub aux.ads
```

This generates the file aux.adb with the following contents:

```
pragma Ada_2012;
package body Aux is

-----
-- Add_One --
-----

function Add_One (V : Integer) return Integer is
begin
    -- Generated stub: replace with real body!
    pragma Compile_Time_Warning (Standard.True, "Add_One unimplemented");
    return raise Program_Error with "Unimplemented function Add_One";
end Add_One;

-----
-- Reset --
-----


procedure Reset (V : in out Integer) is
begin
    -- Generated stub: replace with real body!
    pragma Compile_Time_Warning (Standard.True, "Reset unimplemented");
    raise Program_Error with "Unimplemented procedure Reset";
end Reset;

end Aux;
```

As we can see in this example, not only has **gnatstub** created a package body from all the elements in the package specification, but it also created:

- Headers for each subprogram (as comments);
- Pragmas and exceptions that prevent us from using the unimplemented subprograms in our application.

This is a good starting point for the implementation of the body. Please refer to the section on **gnatstub**¹³⁴ of the GNAT User's Guide for a detailed discussion of **gnatstub** and its options.

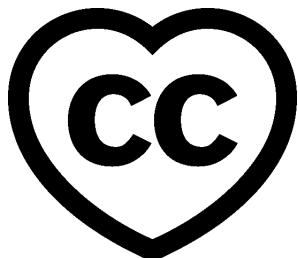
¹³⁴ https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-body-stub-generator-gnatstub

Part IX

Introduction to Ada: Laboratories

Copyright © 2019 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page¹³⁵](#)



These labs contain exercises for the [*Introduction to Ada*](#) (page 5) course.

This document was written by Gustavo A. Hoffmann and reviewed by Michael Frank.

¹³⁵ <http://creativecommons.org/licenses/by-sa/4.0>

IMPERATIVE LANGUAGE

For the exercises below (except for the first one), don't worry about the details of the Main procedure. You should just focus on implementing the application in the subprogram specified by the exercise.

87.1 Hello World

Goal: create a "Hello World!" application.

Steps:

1. Complete the Main procedure.

Requirements:

1. The application must display the message "Hello World!".

Listing 1: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     -- Implement the application here!
6     null;
7 end Main;
```

87.2 Greetings

Goal: create an application that greets a person.

Steps:

1. Complete the Greet procedure.

Requirements:

1. Given an input string <name>, procedure Greet must display the message "Hello <name>!".
 1. For example, if the name is "John", it displays the message "Hello John!".

Remarks:

1. You can use the concatenation operator (&).

Listing 2: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6   procedure Greet (Name : String) is
7   begin
8     -- Implement the application here!
9     null;
10    end Greet;
11
12 begin
13   if Argument_Count < 1 then
14     Put_Line ("ERROR: missing arguments! Exiting...");  

15     return;
16   elsif Argument_Count > 1 then
17     Put_Line ("Ignoring additional arguments...");  

18   end if;
19
20   Greet (Argument (1));
21 end Main;
```

87.3 Positive Or Negative

Goal: create an application that classifies integer numbers.

Steps:

1. Complete the Classify_Number procedure.

Requirements:

1. Given an integer number X, procedure Classify_Number must classify X as positive, negative or zero and display the result:
 1. If $X > 0$, it displays Positive.
 2. If $X < 0$, it displays Negative.
 3. If $X = 0$, it displays Zero.

Listing 3: classify_number.ads

```
1 procedure Classify_Number (X : Integer);
```

Listing 4: classify_number.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5   -- Implement the application here!
6   null;
7 end Classify_Number;
```

Listing 5: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Classify_Number;
5
6  procedure Main is
7    A : Integer;
8  begin
9    if Argument_Count < 1 then
10      Put_Line ("ERROR: missing arguments! Exiting... ");
11      return;
12    elsif Argument_Count > 1 then
13      Put_Line ("Ignoring additional arguments... ");
14    end if;
15
16    A := Integer'Value (Argument (1));
17
18    Classify_Number (A);
19 end Main;

```

87.4 Numbers

Goal: create an application that displays numbers in a specific order.

Steps:

1. Complete the Display_Numbers procedure.

Requirements:

1. Given two integer numbers, Display_Numbers displays all numbers in the range starting with the smallest number.

Listing 6: display_numbers.ads

```

1  procedure Display_Numbers (A, B : Integer);

```

Listing 7: display_numbers.adb

```

1  procedure Display_Numbers (A, B : Integer) is
2  begin
3    -- Implement the application here!
4    null;
5  end Display_Numbers;

```

Listing 8: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Display_Numbers;
5
6  procedure Main is
7    A, B : Integer;
8  begin
9    if Argument_Count < 2 then
10      Put_Line ("ERROR: missing arguments! Exiting... ");

```

(continues on next page)

(continued from previous page)

```
11      return;
12 elsif Argument_Count > 2 then
13   Put_Line ("Ignoring additional arguments..."); 
14 end if;
15
16 A := Integer'Value (Argument (1));
17 B := Integer'Value (Argument (2));
18
19 Display_Numbers (A, B);
20 end Main;
```

SUBPROGRAMS

88.1 Subtract procedure

Goal: write a procedure that subtracts two numbers.

Steps:

1. Complete the procedure Subtract.

Requirements:

1. Subtract performs the operation A - B.

Listing 1: subtract.ads

```
1 -- Write the correct parameters for the procedure below.  
2 procedure Subtract;
```

Listing 2: subtract.adb

```
1 procedure Subtract is  
2 begin  
3     -- Implement the procedure here.  
4     null;  
5 end Subtract;
```

Listing 3: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;  
2 with Ada.Text_Io;          use Ada.Text_Io;  
3  
4 with Subtract;  
5  
procedure Main is  
7     type Test_Case_Index is  
8         (Sub_10_1_Chk,  
9          Sub_10_100_Chk,  
10         Sub_0_5_Chk,  
11         Sub_0_Minus_5_Chk);  
12  
13     procedure Check (TC : Test_Case_Index) is  
14         Result : Integer;  
15     begin  
16         case TC is  
17             when Sub_10_1_Chk =>  
18                 Subtract (10, 1, Result);  
19                 Put_Line ("Result: " & Integer'Image (Result));  
20             when Sub_10_100_Chk =>  
21                 Subtract (10, 100, Result);
```

(continues on next page)

(continued from previous page)

```

22     Put_Line ("Result: " & Integer'Image (Result));
23 when Sub_0_5_Chk =>
24     Subtract (0, 5, Result);
25     Put_Line ("Result: " & Integer'Image (Result));
26 when Sub_0_Minus_5_Chk =>
27     Subtract (0, -5, Result);
28     Put_Line ("Result: " & Integer'Image (Result));
29 end case;
30 end Check;

31
32 begin
33 if Argument_Count < 1 then
34     Put_Line ("ERROR: missing arguments! Exiting...");  

35     return;
36 elsif Argument_Count > 1 then
37     Put_Line ("Ignoring additional arguments...");  

38 end if;
39
40 Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

88.2 Subtract function

Goal: write a function that subtracts two numbers.

Steps:

1. Rewrite the Subtract procedure from the previous exercise as a function.

Requirements:

1. Subtract performs the operation A - B and returns the result.

Listing 4: subtract.ads

```

1 -- Write the correct signature for the function below.
2 -- Don't forget to replace the keyword "procedure" by "function."
3 procedure Subtract;

```

Listing 5: subtract.adb

```

1 procedure Subtract is
2 begin
3     -- Implement the function here!
4     null;
5 end Subtract;

```

Listing 6: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Subtract;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Sub_10_1_Chk,
9          Sub_10_100_Chk,

```

(continues on next page)

(continued from previous page)

```

10    Sub_0_5_Chk,
11    Sub_0_Minus_5_Chk);

12  procedure Check (TC : Test_Case_Index) is
13    Result : Integer;
14
15  begin
16    case TC is
17      when Sub_10_1_Chk =>
18        Result := Subtract (10, 1);
19        Put_Line ("Result: " & Integer'Image (Result));
20      when Sub_10_100_Chk =>
21        Result := Subtract (10, 100);
22        Put_Line ("Result: " & Integer'Image (Result));
23      when Sub_0_5_Chk =>
24        Result := Subtract (0, 5);
25        Put_Line ("Result: " & Integer'Image (Result));
26      when Sub_0_Minus_5_Chk =>
27        Result := Subtract (0, -5);
28        Put_Line ("Result: " & Integer'Image (Result));
29    end case;
30  end Check;

31
32 begin
33  if Argument_Count < 1 then
34    Put_Line ("ERROR: missing arguments! Exiting..."); 
35    return;
36  elsif Argument_Count > 1 then
37    Put_Line ("Ignoring additional arguments..."); 
38  end if;
39
40  Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

88.3 Equality function

Goal: write a function that compares two values and returns a flag.

Steps:

1. Complete the Is_Equal subprogram.

Requirements:

1. Is_Equal returns a flag as a **Boolean** value.
2. The flag must indicate whether the values are equal (flag is **True**) or not (flag is **False**).

Listing 7: is_equal.ads

```

1  -- Write the correct signature for the function below.
2  -- Don't forget to replace the keyword "procedure" by "function."
3  procedure Is_Equal;

```

Listing 8: is_equal.adb

```

1  procedure Is_Equal is
2  begin
3    -- Implement the function here!
4    null;
5  end Is_Equal;

```

Listing 9: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Is_Equal;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Equal_Chk,
9       Inequal_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12
13    procedure Display_Equal (A, B : Integer;
14                             Equal : Boolean) is
15      begin
16        Put (Integer'Image (A));
17        if Equal then
18          Put (" is equal to ");
19        else
20          Put (" isn't equal to ");
21        end if;
22        Put_Line (Integer'Image (B) & ".");
23    end Display_Equal;
24
25    Result : Boolean;
26  begin
27    case TC is
28    when Equal_Chk =>
29      for I in 0 .. 10 loop
30        Result := Is_Equal (I, I);
31        Display_Equal (I, I, Result);
32      end loop;
33    when Inequal_Chk =>
34      for I in 0 .. 10 loop
35        Result := Is_Equal (I, I - 1);
36        Display_Equal (I, I - 1, Result);
37      end loop;
38    end case;
39  end Check;
40
41 begin
42  if Argument_Count < 1 then
43    Put_Line ("ERROR: missing arguments! Exiting...");
44    return;
45  elsif Argument_Count > 1 then
46    Put_Line ("Ignoring additional arguments...");
47  end if;
48
49  Check (Test_Case_Index'Value (Argument (1)));
50 end Main;

```

88.4 States

Goal: write a procedure that displays the state of a machine.

Steps:

1. Complete the procedure `Display_State`.

Requirements:

1. The states can be set according to the following numbers:

Number	State
0	Off
1	On: Simple Processing
2	On: Advanced Processing

2. The procedure `Display_State` receives the number corresponding to a state and displays the state (indicated by the table above) as a user message.

Remarks:

1. You can use a case statement to implement this procedure.

Listing 10: `display_state.ads`

```
1 procedure Display_State (State : Integer);
```

Listing 11: `display_state.adb`

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_State (State : Integer) is
4 begin
5   null;
6 end Display_State;
```

Listing 12: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Display_State;
5
6 procedure Main is
7   State : Integer;
8 begin
9   if Argument_Count < 1 then
10    Put_Line ("ERROR: missing arguments! Exiting...");  

11    return;
12   elsif Argument_Count > 1 then
13    Put_Line ("Ignoring additional arguments...");  

14   end if;
15
16   State := Integer'Value (Argument (1));
17
18   Display_State (State);
19 end Main;
```

88.5 States #2

Goal: write a function that returns the state of a machine.

Steps:

1. Implement the function Get_State.

Requirements:

1. Implement same state machine as in the previous exercise.
2. Function Get_State must return the state as a string.

Remarks:

1. You can implement a function returning a string by simply using quotes in a return statement. For example:

Listing 13: get_hello.ads

```
1 function Get_Hello return String;
```

Listing 14: get_hello.adb

```
1 function Get_Hello return String is
2 begin
3   return "Hello";
4 end Get_Hello;
```

Listing 15: main.adb

```
1 with Ada.Text_Io;      use Ada.Text_Io;
2 with Get_Hello;
3
4 procedure Main is
5   S : constant String := Get_Hello;
6 begin
7   Put_Line (S);
8 end Main;
```

2. You can reuse your previous implementation and replace it by a case expression.

1. For values that do not correspond to a state, you can simply return an empty string ("").

Listing 16: get_state.ads

```
1 function Get_State (State : Integer) return String;
```

Listing 17: get_state.adb

```
1 function Get_State (State : Integer) return String is
2 begin
3   return "";
4 end Get_State;
```

Listing 18: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_Io;      use Ada.Text_Io;
3
```

(continues on next page)

(continued from previous page)

```

4  with Get_State;
5
6  procedure Main is
7      State : Integer;
8  begin
9      if Argument_Count < 1 then
10         Put_Line ("ERROR: missing arguments! Exiting...");  

11         return;
12     elsif Argument_Count > 1 then
13         Put_Line ("Ignoring additional arguments...");  

14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Put_Line (Get_State (State));
19  end Main;

```

88.6 States #3

Goal: implement an on/off indicator for a state machine.

Steps:

1. Implement the function `Is_On`.
2. Implement the procedure `Display_On_Off`.

Requirements:

1. Implement same state machine as in the previous exercise.
2. Function `Is_On` returns:
 - `True` if the machine is on;
 - otherwise, it returns `False`.
3. Procedure `Display_On_Off` displays the message
 - "On" if the machine is on, or
 - "Off" otherwise.
4. `Is_On` must be called in the implementation of `Display_On_Off`.

Remarks:

1. You can implement both subprograms using if expressions.

Listing 19: `is_on.ads`

```

1  function Is_On (State : Integer) return Boolean;

```

Listing 20: `is_on.adb`

```

1  function Is_On (State : Integer) return Boolean is
2  begin
3      return False;
4  end Is_On;

```

Listing 21: display_on_off.ads

```
1 procedure Display_On_Off (State : Integer);
```

Listing 22: display_on_off.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 with Is_On;
3
4 procedure Display_On_Off (State : Integer) is
5 begin
6     Put_Line ("");
7 end Display_On_Off;
```

Listing 23: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Display_On_Off;
5 with Is_On;
6
7 procedure Main is
8     State : Integer;
9 begin
10    if Argument_Count < 1 then
11        Put_Line ("ERROR: missing arguments! Exiting...");  
12        return;
13    elsif Argument_Count > 1 then
14        Put_Line ("Ignoring additional arguments...");  
15    end if;
16
17    State := Integer'Value (Argument (1));
18
19    Display_On_Off (State);
20    Put_Line (Boolean'Image (Is_On (State)));
21 end Main;
```

88.7 States #4

Goal: implement a procedure to update the state of a machine.

Steps:

1. Implement the procedure Set_Next.

Requirements:

1. Implement the same state machine as in the previous exercise.
2. Procedure Set_Next updates the machine's state with the next one in a *circular* manner:
 - In most cases, the next state of N is simply the next number (N + 1).
 - However, if the state is the last one (which is 2 for our machine), the next state must be the first one (in our case: 0).

Remarks:

1. You can use an if expression to implement Set_Next.

Listing 24: set_next.ads

```
1 procedure Set_Next (State : in out Integer);
```

Listing 25: set_next.adb

```
1 procedure Set_Next (State : in out Integer) is
2 begin
3     null;
4 end Set_Next;
```

Listing 26: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Set_Next;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10        Put_Line ("ERROR: missing arguments! Exiting... ");
11        return;
12    elsif Argument_Count > 1 then
13        Put_Line ("Ignoring additional arguments... ");
14    end if;
15
16    State := Integer'Value (Argument (1));
17
18    Set_Next (State);
19    Put_Line (Integer'Image (State));
20 end Main;
```


MODULAR PROGRAMMING

89.1 Months

Goal: create a package to display the months of the year.

Steps:

1. Convert the Months procedure below to a package.
2. Create the specification and body of the Months package.

Requirements:

1. Months must contain the declaration of strings for each month of the year, which are stored in three-character constants based on the month's name.
 - For example, the string "January" is stored in the constant Jan. These strings are then used by the Display_Months procedure, which is also part of the Months package.

Remarks:

1. The goal of this exercise is to create the Months package.
 1. In the code below, Months is declared as a procedure.
 - Therefore, we need to *convert* it into a real package.
2. You have to modify the procedure declaration and implementation in the code below, so that it becomes a package specification and a package body.

Listing 1: months.ads

```
1 -- Create specification for Months package, which includes
2 -- the declaration of the Display_Months procedure.
3 --
4 procedure Months;
```

Listing 2: months.adb

```
1 -- Create body of Months package, which includes
2 -- the implementation of the Display_Months procedure.
3 --
4 procedure Months is
5
6     procedure Display_Months is
7         begin
8             Put_Line ("Months:");
9             Put_Line ("- " & Jan);
10            Put_Line ("- " & Feb);
11            Put_Line ("- " & Mar);
```

(continues on next page)

(continued from previous page)

```

12    Put_Line (" - " & Apr);
13    Put_Line (" - " & May);
14    Put_Line (" - " & Jun);
15    Put_Line (" - " & Jul);
16    Put_Line (" - " & Aug);
17    Put_Line (" - " & Sep);
18    Put_Line (" - " & Oct);
19    Put_Line (" - " & Nov);
20    Put_Line (" - " & Dec);
21 end Display_Months;
22
23 begin
24   null;
25 end Months;

```

Listing 3: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Months;          use Months;
5
6  procedure Main is
7
8    type Test_Case_Index is
9      (Months_Chk);
10
11   procedure Check (TC : Test_Case_Index) is
12   begin
13     case TC is
14       when Months_Chk =>
15         Display_Months;
16     end case;
17   end Check;
18
19 begin
20   if Argument_Count < 1 then
21     Put_Line ("ERROR: missing arguments! Exiting... ");
22     return;
23   elsif Argument_Count > 1 then
24     Put_Line ("Ignoring additional arguments... ");
25   end if;
26
27   Check (Test_Case_Index'Value (Argument (1)));
28 end Main;

```

89.2 Operations

Goal: create a package to perform basic mathematical operations.

Steps:

1. Implement the Operations package.
 1. Declare and implement the Add function.
 2. Declare and implement the Subtract function.
 3. Declare and implement the Multiply: function.

4. Declare and implement the Divide function.
2. Implement the Operations.Test package
 1. Declare and implement the Display procedure.

Requirements:

1. Package Operations contains functions for each of the four basic mathematical operations for parameters of **Integer** type:
 1. Function Add performs the addition of A and B and returns the result;
 2. Function Subtract performs the subtraction of A and B and returns the result;
 3. Function Multiply performs the multiplication of A and B and returns the result;
 4. Function Divide performs the division of A and B and returns the result.
2. Package Operations.Test contains the test environment:
 1. Procedure Display must use the functions from the parent (Operations) package as indicated by the template in the code below.

Listing 4: operations.ads

```

1 package Operations is
2
3   -- Create specification for Operations package, including the
4   -- declaration of the functions mentioned above.
5
6
7 end Operations;

```

Listing 5: operations.adb

```

1 package body Operations is
2
3   -- Create body of Operations package.
4
5
6 end Operations;

```

Listing 6: operations-test.ads

```

1 package Operations.Test is
2
3   -- Create specification for Operations package, including the
4   -- declaration of the Display procedure:
5
6   -- procedure Display (A, B : Integer);
7
8
9 end Operations.Test;

```

Listing 7: operations-test.adb

```

1 package body Operations.Test is
2
3   -- Implement body of Operations.Test package.
4
5
6   procedure Display (A, B : Integer) is
7     A_Str : constant String := Integer'Image (A);
8     B_Str : constant String := Integer'Image (B);

```

(continues on next page)

(continued from previous page)

```

9   begin
10    Put_Line ("Operations:");
11    Put_Line (A_Str & " + " & B_Str & " = "
12              & Integer'Image (Add (A, B))
13              & ",");
14    -- Use the line above as a template and add the rest of the
15    -- implementation for Subtract, Multiply and Divide.
16  end Display;
17
18 end Operations.Test;

```

Listing 8: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Operations;
5  with Operations.Test; use Operations.Test;
6
7  procedure Main is
8
9    type Test_Case_Index is
10      (Operations_Chk,
11       Operations_Display_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14  begin
15    case TC is
16      when Operations_Chk =>
17          Put_Line ("Add (100, 2) = "
18                    & Integer'Image (Operations.Add (100, 2)));
19          Put_Line ("Subtract (100, 2) = "
20                    & Integer'Image (Operations.Subtract (100, 2)));
21          Put_Line ("Multiply (100, 2) = "
22                    & Integer'Image (Operations.Multiply (100, 2)));
23          Put_Line ("Divide (100, 2) = "
24                    & Integer'Image (Operations.Divide (100, 2)));
25      when Operations_Display_Chk =>
26          Display (10, 5);
27          Display (1, 2);
28    end case;
29  end Check;
30
31 begin
32  if Argument_Count < 1 then
33    Put_Line ("ERROR: missing arguments! Exiting... ");
34    return;
35  elsif Argument_Count > 1 then
36    Put_Line ("Ignoring additional arguments... ");
37  end if;
38
39  Check (Test_Case_Index'Value (Argument (1)));
40 end Main;

```

STRONGLY TYPED LANGUAGE

90.1 Colors

Goal: create a package to represent HTML colors in hexadecimal form and its corresponding names.

Steps:

1. Implement the Color_Types package.
 1. Declare the HTML_Color enumeration.
 2. Declare the Basic_HTML_Color enumeration.
 3. Implement the To_Integer function.
 4. Implement the To_HTML_Color function.

Requirements:

1. Enumeration HTML_Color has the following colors:
 - Salmon
 - Firebrick
 - Red
 - Darkred
 - Lime
 - Forestgreen
 - Green
 - Darkgreen
 - Blue
 - Mediumblue
 - Darkblue
2. Enumeration Basic_HTML_Color has the following colors: Red, Green, Blue.
3. Function To_Integer converts from the HTML_Color type to the HTML color code — as integer values in hexadecimal notation.
 - You can find the HTML color codes in the table below.
4. Function To_HTML_Color converts from Basic_HTML_Color to HTML_Color.
5. This is the table to convert from an HTML color to a HTML color code in hexadeciml notation:

Color	HTML color code (hexa)
Salmon	#FA8072
Firebrick	#B22222
Red	#FF0000
Darkred	#8B0000
Lime	#00FF00
Forestgreen	#228B22
Green	#008000
Darkgreen	#006400
Blue	#0000FF
Mediumblue	#0000CD
Darkblue	#00008B

Remarks:

1. In order to express the hexadecimal values above in Ada, use the following syntax:
16#<hex_value># (e.g.: 16#FFFFFF#).
2. For function To_Integer, you may use a **case** for this.

Listing 1: color_types.ads

```

1 package Color_Types is
2
3     -- Include type declaration for HTML_Color!
4
5     -- type HTML_Color is [...]
6
7
8     -- Include function declaration for:
9     -- function To_Integer (C : HTML_Color) return Integer;
10
11    -- Include type declaration for Basic_HTML_Color!
12
13    -- type Basic_HTML_Color is [...]
14
15
16    -- Include function declaration for:
17    -- - Basic_HTML_Color => HTML_Color
18
19    -- function To_HTML_Color [...];
20
21 end Color_Types;

```

Listing 2: color_types.adb

```

1 package body Color_Types is
2
3     -- Implement the conversion from HTML_Color to Integer here!
4
5     -- function To_Integer (C : HTML_Color) return Integer is
6     begin
7         -- Hint: use 'case' for the HTML colors;
8         --       use 16#...# for the hexadecimal values.
9     end To_Integer;
10
11    -- Implement the conversion from Basic_HTML_Color to HTML_Color here!
12
13    -- function To_HTML_Color [...] is
14
15 end Color_Types;

```

Listing 3: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Ada.Integer_Text_IO;
4
5  with Color_Types; use Color_Types;
6
7  procedure Main is
8    type Test_Case_Index is
9      (HTML_Color_Range,
10       HTML_Color_To_Integer,
11       Basic_HTML_Color_To_HTML_Color);
12
13  procedure Check (TC : Test_Case_Index) is
14  begin
15    case TC is
16      when HTML_Color_Range =>
17          for I in HTML_Color'Range loop
18              Put_Line (HTML_Color'Image (I));
19          end loop;
20      when HTML_Color_To_Integer =>
21          for I in HTML_Color'Range loop
22              Ada.Integer_Text_IO.Put (Item  => To_Integer (I),
23                                      Width => 6,
24                                      Base   => 16);
25              New_Line;
26          end loop;
27      when Basic_HTML_Color_To_HTML_Color =>
28          for I in Basic_HTML_Color'Range loop
29              Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30          end loop;
31    end case;
32  end Check;
33
34 begin
35  if Argument_Count < 1 then
36    Put_Line ("ERROR: missing arguments! Exiting...");  

37    return;
38  elsif Argument_Count > 1 then
39    Put_Line ("Ignoring additional arguments...");  

40  end if;
41
42  Check (Test_Case_Index'Value (Argument (1)));
43 end Main;

```

90.2 Integers

Goal: implement a package with various integer types.

Steps:

1. Implement the Int_Types package.
 1. Declare the integer type I_100.
 2. Declare the modular type U_100.
 3. Implement the To_I_100 function to convert from the U_100 type.
 4. Implement the To_U_100 function to convert from the I_100 type.

5. Declare the derived type D_50.
6. Declare the subtype S_50.
7. Implement the To_D_50 function to convert from the I_100 type.
8. Implement the To_S_50 function to convert from the I_100 type.
9. Implement the To_I_100 function to convert from the D_50 type.

Requirements:

1. Types I_100 and U_100 have values between 0 and 100.
 1. Type I_100 is an integer type.
 2. Type U_100 is a modular type.
2. Function To_I_100 converts from the U_100 type to the I_100 type.
3. Function To_U_100 converts from the I_100 type to the U_100 type.
4. Types D_50 and S_50 have values between 10 and 50 and use I_100 as a base type.
 1. D_50 is a derived type.
 2. S_50 is a subtype.
5. Function To_D_50 converts from the I_100 type to the D_50 type.
6. Function To_S_50 converts from the I_100 type to the S_50 type.
7. Functions To_D_50 and To_S_50 saturate the input values if they are out of range.
 - If the input is less than 10 the output should be 10.
 - If the input is greater than 50 the output should be 50.
8. Function To_I_100 converts from the D_50 type to the I_100 type.

Remarks:

1. For the implementation of functions To_D_50 and To_S_50, you may use the type attributes D_50'First and D_50'Last:
 1. D_50'First indicates the minimum value of the D_50 type.
 2. D_50>Last indicates the maximum value of the D_50 type.
 3. The same attributes are available for the S_50 type (S_50'First and S_50>Last).
2. We could have implemented a function To_I_100 as well to convert from S_50 to I_100. However, we skip this here because explicit conversions are not needed for subtypes.

Listing 4: int_types.ads

```
1 package Int_Types is
2
3   -- Include type declarations for I_100 and U_100!
4
5   -- type I_100 is [...]
6   -- type U_100 is [...]
7
8
9   function To_I_100 (V : U_100) return I_100;
10
11  function To_U_100 (V : I_100) return U_100;
12
13  -- Include type declarations for D_50 and S_50!
14
```

(continues on next page)

(continued from previous page)

```

15  -- [...] D_50 is [...]
16  -- [...] S_50 is [...]
17  --
18
19  function To_D_50 (V : I_100) return D_50;
20
21  function To_S_50 (V : I_100) return S_50;
22
23  function To_I_100 (V : D_50) return I_100;
24
25 end Int_Types;

```

Listing 5: int_types.adb

```

1 package body Int_Types is
2
3     function To_I_100 (V : U_100) return I_100 is
4     begin
5         -- Implement the conversion from U_100 to I_100 here!
6         --
7         null;
8     end To_I_100;
9
10    function To_U_100 (V : I_100) return U_100 is
11    begin
12        -- Implement the conversion from I_100 to U_100 here!
13        --
14        null;
15    end To_U_100;
16
17    function To_D_50 (V : I_100) return D_50 is
18        Min : constant I_100 := I_100 (D_50'First);
19        Max : constant I_100 := I_100 (D_50'Last);
20    begin
21        -- Implement the conversion from I_100 to D_50 here!
22        --
23        -- Hint: using the constants above simplifies the checks needed for
24        -- this function.
25        --
26        null;
27    end To_D_50;
28
29    function To_S_50 (V : I_100) return S_50 is
30    begin
31        -- Implement the conversion from I_100 to S_50 here!
32        --
33        -- Remark: don't forget to verify whether an explicit conversion like
34        --          S_50 (V) is needed.
35        --
36        null;
37    end To_S_50;
38
39    function To_I_100 (V : D_50) return I_100 is
40    begin
41        -- Implement the conversion from I_100 to D_50 here!
42        --
43        -- Remark: don't forget to verify whether an explicit conversion like
44        --          I_100 (V) is needed.
45        --
46        null;
47    end To_I_100;

```

(continues on next page)

(continued from previous page)

```
48
49 end Int_Types;
```

Listing 6: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Int_Types;        use Int_Types;
5
6  procedure Main is
7    package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
8    package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
9    package D_50_IO  is new Ada.Text_IO.Integer_IO (D_50);
10
11   use I_100_IO;
12   use U_100_IO;
13   use D_50_IO;
14
15  type Test_Case_Index is
16    (I_100_Range,
17     U_100_Range,
18     U_100_Wraparound,
19     U_100_To_I_100,
20     I_100_To_U_100,
21     D_50_Range,
22     S_50_Range,
23     I_100_To_D_50,
24     I_100_To_S_50,
25     D_50_To_I_100,
26     S_50_To_I_100);
27
28  procedure Check (TC : Test_Case_Index) is
29 begin
30   I_100_IO.Default_Width := 1;
31   U_100_IO.Default_Width := 1;
32   D_50_IO.Default_Width := 1;
33
34  case TC is
35    when I_100_Range =>
36      Put (I_100'First);
37      New_Line;
38      Put (I_100'Last);
39      New_Line;
40    when U_100_Range =>
41      Put (U_100'First);
42      New_Line;
43      Put (U_100'Last);
44      New_Line;
45    when U_100_Wraparound =>
46      Put (U_100'First - 1);
47      New_Line;
48      Put (U_100'Last + 1);
49      New_Line;
50    when U_100_To_I_100 =>
51      for I in U_100'Range loop
52        I_100_IO.Put (To_I_100 (I));
53        New_Line;
54      end loop;
55    when I_100_To_U_100 =>
56      for I in I_100'Range loop
```

(continues on next page)

(continued from previous page)

```

57      Put (To_U_100 (I));
58      New_Line;
59  end loop;
60 when D_50_Range =>
61   Put (D_50'First);
62   New_Line;
63   Put (D_50'Last);
64   New_Line;
65 when S_50_Range =>
66   Put (S_50'First);
67   New_Line;
68   Put (S_50'Last);
69   New_Line;
70 when I_100_To_D_50 =>
71   for I in I_100'Range loop
72     Put (To_D_50 (I));
73     New_Line;
74   end loop;
75 when I_100_To_S_50 =>
76   for I in I_100'Range loop
77     Put (To_S_50 (I));
78     New_Line;
79   end loop;
80 when D_50_To_I_100 =>
81   for I in D_50'Range loop
82     Put (To_I_100 (I));
83     New_Line;
84   end loop;
85 when S_50_To_I_100 =>
86   for I in S_50'Range loop
87     Put (I);
88     New_Line;
89   end loop;
90 end case;
91 end Check;

92
93 begin
94  if Argument_Count < 1 then
95    Put_Line ("ERROR: missing arguments! Exiting...");
96    return;
97  elsif Argument_Count > 1 then
98    Put_Line ("Ignoring additional arguments...");
99  end if;
100
101 Check (Test_Case_Index'Value (Argument (1)));
102 end Main;

```

90.3 Temperatures

Goal: create a package to handle temperatures in Celsius and Kelvin.

Steps:

1. Implement the Temperature_Types package.
 1. Declare the Celsius type.
 2. Declare the Int_Celsius type.
 3. Implement the To_Celsius function.

4. Implement the `To_Int_Celsius` function.
5. Declare the `Kelvin` type.
6. Implement the `To_Celsius` function to convert from the `Kelvin` type.
7. Implement the `To_Kelvin` function.

Requirements:

1. The custom floating-point types declared in `Temperature_Types` must use a precision of six digits.
2. Types `Celsius` and `Int_Celsius` are used for temperatures in Celsius:
 1. `Celsius` is a floating-point type with a range between -273.15 and 5504.85.
 2. `Int_Celsius` is an integer type with a range between -273 and 5505.
3. Functions `To_Celsius` and `To_Int_Celsius` are used for type conversion:
 1. `To_Celsius` converts from `Int_Celsius` to `Celsius` type.
 2. `To_Int_Celsius` converts from `Celsius` and `Int_Celsius` types.
4. `Kelvin` is a floating-point type for temperatures in Kelvin using a range between 0.0 and 5778.0.
5. The functions `To_Celsius` and `To_Kelvin` are used to convert between temperatures in Kelvin and Celsius.
 1. In order to convert temperatures in Celsius to Kelvin, you must use the formula $K = C + 273.15$, where:
 - K is the temperature in Kelvin, and
 - C is the temperature in Celsius.

Remarks:

1. When implementing the `To_Celsius` function for the `Int_Celsius` type:
 1. You'll need to check for the minimum and maximum values of the input values because of the slightly different ranges.
 2. You may use variables of floating-point type (**Float**) for intermediate values.
2. For the implementation of the functions `To_Celsius` and `To_Kelvin` (used for converting between Kelvin and Celsius), you may use a variable of floating-point type (**Float**) for intermediate values.

Listing 7: `temperature_types.ads`

```
1 package Temperature_Types is
2
3     -- Include type declaration for Celsius!
4
5     -- Celsius is [...];
6     -- Int_Celsius is [...];
7
8
9     function To_Celsius (T : Int_Celsius) return Celsius;
10
11    function To_Int_Celsius (T : Celsius) return Int_Celsius;
12
13    -- Include type declaration for Kelvin!
14
15    -- type Kelvin is [...];
16
```

(continues on next page)

(continued from previous page)

```

17
18      -- Include function declarations for:
19      -- - Kelvin => Celsius
20      -- - Celsius => Kelvin
21
22      --
23      -- function To_Celsius [...];
24      -- function To_Kelvin [...];
25
26 end Temperature_Types;

```

Listing 8: temperature_types.adb

```

1 package body Temperature_Types is
2
3     function To_Celsius (T : Int_Celsius) return Celsius is
4     begin
5         null;
6     end To_Celsius;
7
8     function To_Int_Celsius (T : Celsius) return Int_Celsius is
9     begin
10        null;
11    end To_Int_Celsius;
12
13     -- Include function implementation for:
14     -- - Kelvin => Celsius
15     -- - Celsius => Kelvin
16
17     --
18     -- function To_Celsius [...] is
19     -- function To_Kelvin [...] is
20
21 end Temperature_Types;

```

Listing 9: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO; use Ada.Text_IO;
3
4 with Temperature_Types; use Temperature_Types;
5
6 procedure Main is
7     package Celsius_I0 is new Ada.Text_IO.Float_IO (Celsius);
8     package Kelvin_I0 is new Ada.Text_IO.Float_IO (Kelvin);
9     package Int_Celsius_I0 is new Ada.Text_IO.Integer_IO (Int_Celsius);
10
11    use Celsius_I0;
12    use Kelvin_I0;
13    use Int_Celsius_I0;
14
15    type Test_Case_Index is
16        (Celsius_Range,
17         Celsius_To_Int_Celsius,
18         Int_Celsius_To_Celsius,
19         Kelvin_To_Celsius,
20         Celsius_To_Kelvin);
21
22    procedure Check (TC : Test_Case_Index) is
23    begin
24        Celsius_I0.Default_Fore := 1;
25        Kelvin_I0.Default_Fore := 1;

```

(continues on next page)

(continued from previous page)

```

26     Int_Celsius_IO.Default_Width := 1;
27
28 case TC is
29     when Celsius_Range =>
30         Put (Celsius'First);
31         New_Line;
32         Put (Celsius'Last);
33         New_Line;
34     when Celsius_To_Int_Celsius =>
35         Put (To_Int_Celsius (Celsius'First));
36         New_Line;
37         Put (To_Int_Celsius (0.0));
38         New_Line;
39         Put (To_Int_Celsius (Celsius'Last));
40         New_Line;
41     when Int_Celsius_To_Celsius =>
42         Put (To_Celsius (Int_Celsius'First));
43         New_Line;
44         Put (To_Celsius (0));
45         New_Line;
46         Put (To_Celsius (Int_Celsius'Last));
47         New_Line;
48     when Kelvin_To_Celsius =>
49         Put (To_Celsius (Kelvin'First));
50         New_Line;
51         Put (To_Celsius (0));
52         New_Line;
53         Put (To_Celsius (Kelvin'Last));
54         New_Line;
55     when Celsius_To_Kelvin =>
56         Put (To_Kelvin (Celsius'First));
57         New_Line;
58         Put (To_Kelvin (Celsius'Last));
59         New_Line;
60 end case;
61 end Check;
62
63 begin
64     if Argument_Count < 1 then
65         Put_Line ("ERROR: missing arguments! Exiting...");
66         return;
67     elsif Argument_Count > 1 then
68         Put_Line ("Ignoring additional arguments...");
69     end if;
70
71     Check (Test_Case_Index'Value (Argument (1)));
72 end Main;

```

RECORDS

91.1 Directions

Goal: create a package that handles directions and geometric angles.

Steps:

1. Implement the Directions package.
 1. Declare the Ext_Angle record.
 2. Implement the Display procedure.
 3. Implement the To_Ext_Angle function.

Requirements:

1. Record Ext_Angle stores information about the extended angle (see remark about *extended angles* below).
2. Procedure Display displays information about the extended angle.
 1. You should use the implementation that has been commented out (see code below) as a starting point.
3. Function To_Ext_Angle converts a simple angle value to an extended angle (Ext_Angle type).

Remarks:

1. We make use of the algorithm implemented in the Check_Direction procedure (*chapter on imperative language* (page 9)).
2. For the sake of this exercise, we use the concept of *extended angles*. This includes the actual geometric angle and the corresponding direction (North, South, Northwest, and so on).

Listing 1: directions.ads

```
1 package Directions is
2
3     type Angle_Mod is mod 360;
4
5     type Direction is
6         (North,
7          Northeast,
8          East,
9          Southeast,
10         South,
11         Southwest,
12         West,
13         Northwest);
```

(continues on next page)

(continued from previous page)

```

14
15  function To_Direction (N: Angle_Mod) return Direction;
16
17  -- Include type declaration for Ext_Angle record type:
18
19  -- NOTE: Use the Angle_Mod and Direction types declared above!
20
21  -- type Ext_Angle is [...]
22
23
24  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
25
26  procedure Display (N : Ext_Angle);
27
28 end Directions;
```

Listing 2: directions.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Directions is
4
5  procedure Display (N : Ext_Angle) is
6  begin
7    -- Uncomment the code below and fill the missing elements
8
9    -- Put_Line ("Angle: "
10    --            & Angle_Mod'Image (____)
11    --            & " => "
12    --            & Direction'Image (____)
13    --            & ".");
14    null;
15  end Display;
16
17  function To_Direction (N : Angle_Mod) return Direction is
18  begin
19    case N is
20      when 0          => return North;
21      when 1 .. 89   => return Northeast;
22      when 90        => return East;
23      when 91 .. 179 => return Southeast;
24      when 180       => return South;
25      when 181 .. 269 => return Southwest;
26      when 270       => return West;
27      when 271 .. 359 => return Northwest;
28    end case;
29  end To_Direction;
30
31  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
32  begin
33    -- Implement the conversion from Angle_Mod to Ext_Angle here!
34
35    -- Hint: you can use a return statement and an aggregate.
36
37    null;
38  end To_Ext_Angle;
39
40 end Directions;
```

Listing 3: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Directions;         use Directions;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Direction_Chk);
9
10   procedure Check (TC : Test_Case_Index) is
11   begin
12     case TC is
13       when Direction_Chk =>
14         Display (To_Ext_Angle (0));
15         Display (To_Ext_Angle (30));
16         Display (To_Ext_Angle (45));
17         Display (To_Ext_Angle (90));
18         Display (To_Ext_Angle (91));
19         Display (To_Ext_Angle (120));
20         Display (To_Ext_Angle (180));
21         Display (To_Ext_Angle (250));
22         Display (To_Ext_Angle (270));
23     end case;
24   end Check;
25
26 begin
27   if Argument_Count < 1 then
28     Put_Line ("ERROR: missing arguments! Exiting...");
29     return;
30   elsif Argument_Count > 1 then
31     Put_Line ("Ignoring additional arguments...");
32   end if;
33
34   Check (Test_Case_Index'Value (Argument (1)));
35 end Main;

```

91.2 Colors

Goal: create a package to represent HTML colors in RGB format using the hexadecimal form.

Steps:

1. Implement the Color_Types package.
 1. Declare the RGB record.
 2. Implement the To_RGB function.
 3. Implement the Image function for the RGB type.

Requirements:

1. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

Color	Red	Green	Blue
Salmon	#FA	#80	#72
Firebrick	#B2	#22	#22
Red	#FF	#00	#00
Darkred	#8B	#00	#00
Lime	#00	#FF	#00
Forestgreen	#22	#8B	#22
Green	#00	#80	#00
Darkgreen	#00	#64	#00
Blue	#00	#00	#FF
Mediumblue	#00	#00	#CD
Darkblue	#00	#00	#8B

2. The hexadecimal information of each HTML color can be mapped to three color elements: red, green and blue.
 1. Each color element has a value between 0 and 255, or 00 and FF in hexadecimal.
 2. For example, for the color *salmon*, the hexadecimal value of the color elements are:
 - red = FA,
 - green = 80, and
 - blue = 72.
3. Record RGB stores information about HTML colors in RGB format, so that we can retrieve the individual color elements.
4. Function To_RGB converts from the HTML_Color enumeration to the RGB type based on the information from the table above.
5. Function Image returns a string representation of the RGB type in this format:
 - "(Red => 16#..#, Green => 16#...#, Blue => 16#...#)"

Remarks:

1. We use the exercise on HTML colors from the previous lab on *Strongly typed language* (page 987) as a starting point.

Listing 4: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   function To_Integer (C : HTML_Color) return Integer;
17
18   type Basic_HTML_Color is
19     (Red,

```

(continues on next page)

(continued from previous page)

```

20      Green,
21      Blue);
22
23  function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25  subtype Int_Color is Integer range 0 .. 255;
26
27  -- Replace type declaration for RGB record below
28  --
29  -- - NOTE: Use the Int_Color type declared above!
30  --
31  -- type RGB is [...]
32  --
33  type RGB is null record;
34
35  function To_RGB (C : HTML_Color) return RGB;
36
37  function Image (C : RGB) return String;
38
39 end Color_Types;

```

Listing 5: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5    function To_Integer (C : HTML_Color) return Integer is
6    begin
7      case C is
8        when Salmon      => return 16#FA8072#;
9        when Firebrick   => return 16#B22222#;
10       when Red         => return 16#FF0000#;
11       when Darkred    => return 16#8B0000#;
12       when Lime        => return 16#00FF00#;
13       when Forestgreen => return 16#228B22#;
14       when Green       => return 16#008000#;
15       when Darkgreen   => return 16#006400#;
16       when Blue        => return 16#0000FF#;
17       when Mediumblue  => return 16#0000CD#;
18       when Darkblue    => return 16#00008B#;
19     end case;
20
21   end To_Integer;
22
23   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
24   begin
25     case C is
26       when Red      => return Red;
27       when Green   => return Green;
28       when Blue    => return Blue;
29     end case;
30   end To_HTML_Color;
31
32   function To_RGB (C : HTML_Color) return RGB is
33   begin
34     -- Implement the conversion from HTML_Color to RGB here!
35     --
36     return (null record);
37   end To_RGB;
38

```

(continues on next page)

(continued from previous page)

```

39  function Image (C : RGB) return String is
40    subtype Str_Range is Integer range 1 .. 10;
41    SR : String (Str_Range);
42    SG : String (Str_Range);
43    SB : String (Str_Range);
44  begin
45    -- Replace argument in the calls to Put below
46    -- with the missing elements (red, green, blue)
47    -- from the RGB record
48    --
49    Ada.Integer_Text_Io.Put (To      => SR,
50                           Item    => 0,      -- REPLACE!
51                           Base    => 16);
52    Ada.Integer_Text_Io.Put (To      => SG,
53                           Item    => 0,      -- REPLACE!
54                           Base    => 16);
55    Ada.Integer_Text_Io.Put (To      => SB,
56                           Item    => 0,      -- REPLACE!
57                           Base    => 16);
58    return ("(Red => " & SR
59           & ", Green => " & SG
60           & ", Blue => " & SB
61           &"')");
62  end Image;
63
64 end Color_Types;

```

Listing 6: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_Io;       use Ada.Text_Io;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7    type Test_Case_Index is
8      (HTML_Color_To_RGB);
9
10   procedure Check (TC : Test_Case_Index) is
11   begin
12     case TC is
13       when HTML_Color_To_RGB =>
14         for I in HTML_Color'Range loop
15           Put_Line (HTML_Color'Image (I) & " => "
16                     & Image (To_RGB (I)) & ".");
17         end loop;
18     end case;
19   end Check;
20
21 begin
22   if Argument_Count < 1 then
23     Put_Line ("ERROR: missing arguments! Exiting... ");
24     return;
25   elsif Argument_Count > 1 then
26     Put_Line ("Ignoring additional arguments... ");
27   end if;
28
29   Check (Test_Case_Index'Value (Argument (1)));
30 end Main;

```

91.3 Inventory

Goal: create a simplified inventory system for a store to enter items and keep track of assets.

Steps:

1. Implement the `Inventory_Pkg` package.
 1. Declare the `Item` record.
 2. Implement the `Init` function.
 3. Implement the `Add` procedure.

Requirements:

1. Record `Item` collects information about products from the store.
 1. To keep it simple, this record only contains the name, quantity and price of each item.
 2. The record components are:
 - Name of `Item_Name` type;
 - Quantity of `Natural` type;
 - Price of `Float` type.
2. Function `Init` returns an initialized item (of `Item` type).
 1. Function `Init` must also display the item name by calling the `To_String` function for the `Item_Name` type.
 - This is already implemented in the code below.
3. Procedure `Add` adds an item to the assets.
 1. Since we want to keep track of the assets, the implementation must accumulate the total value of each item's inventory, the result of multiplying the item quantity and its price.

Listing 7: `inventory_pkg.ads`

```

1  package Inventory_Pkg is
2
3    type Item_Name is
4      (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);
5
6    function To_String (I : Item_Name) return String;
7
8    -- Replace type declaration for Item record:
9
10   type Item is null record;
11
12   function Init (Name      : Item_Name;
13                  Quantity  : Natural;
14                  Price     : Float) return Item;
15
16   procedure Add (Assets : in out Float;
17                  I       : Item);
18
19 end Inventory_Pkg;

```

Listing 8: inventory_pkg.adb

```

1  with Ada.Text_Io; use Ada.Text_Io;
2
3  package body Inventory_Pkg is
4
5      function To_String (I : Item_Name) return String is
6  begin
7      case I is
8          when Ballpoint_Pen      => return "Ballpoint Pen";
9          when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
10         when Feather_Quill_Pen   => return "Feather Quill Pen";
11     end case;
12  end To_String;
13
14  function Init (Name      : Item_Name;
15                  Quantity   : Natural;
16                  Price      : Float) return Item is
17  begin
18      Put_Line ("Item: " & To_String (Name) & ".");
19
20      -- Replace return statement with the actual record initialization!
21      --
22      return (null record);
23  end Init;
24
25  procedure Add (Assets : in out Float;
26                  I       : Item) is
27  begin
28      -- Implement the function that adds an item to the inventory here!
29      --
30      null;
31  end Add;
32
33 end Inventory_Pkg;

```

Listing 9: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_Io;        use Ada.Text_Io;
3
4  with Inventory_Pkg;      use Inventory_Pkg;
5
6  procedure Main is
7      -- Remark: the following line is not relevant.
8      F : array (1 .. 10) of Float := (others => 42.42);
9
10 type Test_Case_Index is
11     (Inventory_Chk);
12
13 procedure Display (Assets : Float) is
14     package F_IO is new Ada.Text_Io.Float_Io (Float);
15
16     use F_IO;
17 begin
18     Put ("Assets: $");
19     Put (Assets, 1, 2, 0);
20     Put (".");
21     New_Line;
22 end Display;
23

```

(continues on next page)

(continued from previous page)

```

24  procedure Check (TC : Test_Case_Index) is
25      I      : Item;
26      Assets : Float := 0.0;
27
28      -- Please ignore the following three lines!
29      pragma Warnings (Off, "default initialization");
30      for Assets'Address use F'Address;
31      pragma Warnings (On, "default initialization");
32  begin
33      case TC is
34          when Inventory_Chk =>
35              I := Init (Ballpoint_Pen,           185,  0.15);
36              Add (Assets, I);
37              Display (Assets);
38
39              I := Init (Oil_Based_Pen_Marker, 100,  9.0);
40              Add (Assets, I);
41              Display (Assets);
42
43              I := Init (Feather_Quill_Pen,      2,  40.0);
44              Add (Assets, I);
45              Display (Assets);
46      end case;
47  end Check;
48
49 begin
50     if Argument_Count < 1 then
51         Put_Line ("ERROR: missing arguments! Exiting...");  

52         return;
53     elsif Argument_Count > 1 then
54         Put_Line ("Ignoring additional arguments...");  

55     end if;
56
57     Check (Test_Case_Index'Value (Argument (1)));
58 end Main;

```


ARRAYS

92.1 Constrained Array

Goal: declare a constrained array and implement operations on it.

Steps:

1. Implement the Constrained_Arrays package.
 1. Declare the range type My_Index.
 2. Declare the array type My_Array.
 3. Declare and implement the Init function.
 4. Declare and implement the Double procedure.
 5. Declare and implement the First_Elem function.
 6. Declare and implement the Last_Elem function.
 7. Declare and implement the Length function.
 8. Declare the object A of My_Array type.

Requirements:

1. Range type My_Index has a range from 1 to 10.
2. My_Array is a constrained array of **Integer** type.
 1. It must make use of the My_Index type.
 2. It is therefore limited to 10 elements.
3. Function Init returns an array where each element is initialized with the corresponding index.
4. Procedure Double doubles the value of each element of an array.
5. Function First_Elem returns the first element of the array.
6. Function Last_Elem returns the last element of the array.
7. Function Length returns the length of the array.
8. Object A of My_Array type is initialized with:
 1. the values 1 and 2 for the first two elements, and
 2. 42 for all other elements.

Listing 1: constrained_arrays.ads

```
1 package Constrained_Arrays is
2
3     -- Complete the type and subprogram declarations:
4     --
5     -- type My_Index is [...]
6     --
7     -- type My_Array is [...]
8     --
9     -- function Init ...
10    --
11    -- procedure Double ...
12    --
13    -- function First_Elem ...
14    --
15    -- function Last_Elem ...
16    --
17    -- function Length ...
18    --
19    -- A : ...
20
21 end Constrained_Arrays;
```

Listing 2: constrained_arrays.adb

```
1 package body Constrained_Arrays is
2
3     -- Create the implementation of the subprograms!
4     --
5
6 end Constrained_Arrays;
```

Listing 3: main.adb

```
1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Constrained_Arrays; use Constrained_Arrays;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Range_Chk,
9          Array_Range_Chk,
10         A_Obj_Chk,
11         Init_Chk,
12         Double_Chk,
13         First_Elem_Chk,
14         Last_Elem_Chk,
15         Length_Chk);
16
17 procedure Check (TC : Test_Case_Index) is
18     AA : My_Array;
19
20     procedure Display (A : My_Array) is
21 begin
22     for I in A'Range loop
23         Put_Line (Integer'Image (A (I)));
24     end loop;
25 end Display;
26
27 procedure Local_Init (A : in out My_Array) is
```

(continues on next page)

(continued from previous page)

```

28 begin
29     A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
30 end Local_Init;
31 begin
32     case TC is
33         when Range_Chk =>
34             for I in My_Index loop
35                 Put_Line (My_Index'Image (I));
36             end loop;
37         when Array_Range_Chk =>
38             for I in My_Array'Range loop
39                 Put_Line (My_Index'Image (I));
40             end loop;
41         when A_Obj_Chk =>
42             Display (A);
43         when Init_Chk =>
44             AA := Init;
45             Display (AA);
46         when Double_Chk =>
47             Local_Init (AA);
48             Double (AA);
49             Display (AA);
50         when First_Elem_Chk =>
51             Local_Init (AA);
52             Put_Line (Integer'Image (First_Elem (AA)));
53         when Last_Elem_Chk =>
54             Local_Init (AA);
55             Put_Line (Integer'Image (Last_Elem (AA)));
56         when Length_Chk =>
57             Put_Line (Integer'Image (Length (AA)));
58     end case;
59 end Check;

60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting...");  

63         return;  

64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments...");  

66     end if;  

67  

68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

92.2 Colors: Lookup-Table

Goal: rewrite a package to represent HTML colors in RGB format using a lookup table.

Steps:

1. Implement the Color_Types package.
 1. Declare the array type HTML_Color_RGB.
 2. Declare the To_RGB_Lookup_Table object and initialize it.
 3. Adapt the implementation of the To_RGB function.

Requirements:

1. Array type HTML_Color_RGB is used for the table.

2. The To_RGB_Lookup_Table object of HTML_Color_RGB type contains the lookup table.
 - This table must be implemented as an array of constant values.
3. The implementation of the To_RGB function must use the To_RGB_Lookup_Table object.

Remarks:

1. This exercise is based on the HTML colors exercise from a previous lab (*Records* (page 997)).
2. In the previous implementation, you could use a **case** statement to implement the To_RGB function. Here, you must rewrite the function using a look-up table.
 1. The implementation of the To_RGB function below includes the case statement as commented-out code. You can use this as your starting point: you just need to copy it and convert the case statement to an array declaration.
 1. Don't use a case statement to implement the To_RGB function. Instead, write code that accesses To_RGB_Lookup_Table to get the correct value.
3. The following table contains the HTML colors and the corresponding value in hexadecimal form for each color element:

Color	Red	Green	Blue
Salmon	#FA	#80	#72
Firebrick	#B2	#22	#22
Red	#FF	#00	#00
Darkred	#8B	#00	#00
Lime	#00	#FF	#00
Forestgreen	#22	#8B	#22
Green	#00	#80	#00
Darkgreen	#00	#64	#00
Blue	#00	#00	#FF
Mediumblue	#00	#00	#CD
Darkblue	#00	#00	#8B

Listing 4: color_types.ads

```
1 package Color_Types is
2
3     type HTML_Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     subtype Int_Color is Integer range 0 .. 255;
17
18     type RGB is record
19         Red   : Int_Color;
20         Green : Int_Color;
21         Blue  : Int_Color;
22     end record;
```

(continues on next page)

(continued from previous page)

```

23
24   function To_RGB (C : HTML_Color) return RGB;
25
26   function Image (C : RGB) return String;
27
28   -- Declare array type for lookup table here:
29   --
30   -- type HTML_Color_RGB is ...
31
32   -- Declare lookup table here:
33   --
34   -- To_RGB_Lookup_Table : ...
35
36 end Color_Types;

```

Listing 5: color_types.adb

```

1  with Ada.Integer_Text_Io;
2  package body Color_Types is
3
4    function To_RGB (C : HTML_Color) return RGB is
5    begin
6      -- Implement To_RGB using To_RGB_Lookup_Table
7      return (0, 0, 0);
8
9      -- Use the code below from the previous version of the To_RGB
10     -- function to declare the To_RGB_Lookup_Table:
11     --
12     -- case C is
13     --   when Salmon      => return (16#FA#, 16#80#, 16#72#);
14     --   when Firebrick   => return (16#B2#, 16#22#, 16#22#);
15     --   when Red         => return (16#FF#, 16#00#, 16#00#);
16     --   when Darkred    => return (16#8B#, 16#00#, 16#00#);
17     --   when Lime        => return (16#00#, 16#FF#, 16#00#);
18     --   when Forestgreen => return (16#22#, 16#8B#, 16#22#);
19     --   when Green       => return (16#00#, 16#80#, 16#00#);
20     --   when Darkgreen   => return (16#00#, 16#64#, 16#00#);
21     --   when Blue        => return (16#00#, 16#00#, 16#FF#);
22     --   when Mediumblue  => return (16#00#, 16#00#, 16#CD#);
23     --   when Darkblue    => return (16#00#, 16#00#, 16#8B#);
24   end case;
25
26 end To_RGB;
27
28 function Image (C : RGB) return String is
29   subtype Str_Range is Integer range 1 .. 10;
30   SR : String (Str_Range);
31   SG : String (Str_Range);
32   SB : String (Str_Range);
33 begin
34   Ada.Integer_Text_Io.Put (To      => SR,
35                           Item    => C.Red,
36                           Base    => 16);
37   Ada.Integer_Text_Io.Put (To      => SG,
38                           Item    => C.Green,
39                           Base    => 16);
40   Ada.Integer_Text_Io.Put (To      => SB,
41                           Item    => C.Blue,
42                           Base    => 16);
43   return ("(Red => " & SR
44          & ", Green => " & SG

```

(continues on next page)

(continued from previous page)

```

45      & ", Blue => " & SB
46      &""));
47  end Image;
48
49 end Color_Types;
```

Listing 6: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Color_Types;          use Color_Types;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Color_Table_Chk,
9       HTML_Color_To_Integer_Chk);
10
11 procedure Check (TC : Test_Case_Index) is
12 begin
13   case TC is
14     when Color_Table_Chk =>
15       Put_Line ("Size of HTML_Color_RGB: "
16                  & Integer'Image (HTML_Color_RGB'Length));
17       Put_Line ("Firebrick: "
18                  & Image (To_RGB_Lookup_Table (Firebrick)));
19     when HTML_Color_To_Integer_Chk =>
20       for I in HTML_Color'Range loop
21         Put_Line (HTML_Color'Image (I) & " => "
22                    & Image (To_RGB (I)) & ".");
23       end loop;
24   end case;
25 end Check;
26
27 begin
28   if Argument_Count < 1 then
29     Put_Line ("ERROR: missing arguments! Exiting...");  

30     return;
31   elsif Argument_Count > 1 then
32     Put_Line ("Ignoring additional arguments...");  

33   end if;
34
35   Check (Test_Case_Index'Value (Argument (1)));
36 end Main;
```

92.3 Unconstrained Array

Goal: declare an unconstrained array and implement operations on it.

Steps:

1. Implement the Unconstrained_Arrays package.
 1. Declare the My_Array type.
 2. Declare and implement the Init procedure.
 3. Declare and implement the Init function.
 4. Declare and implement the Double procedure.

5. Declare and implement the Diff_Prev_Elem function.

Requirements:

1. My_Array is an unconstrained array (with a **Positive** range) of **Integer** elements.
2. Procedure Init initializes each element with the index starting with the last one.
 - For example, for an array of 3 elements where the index of the first element is 1 (My_Array (1 .. 3)), the values of these elements after a call to Init must be (3, 2, 1).
3. Function Init returns an array based on the length L and start index I provided to the Init function.
 1. I indicates the index of the first element of the array.
 2. L indicates the length of the array.
 3. Both I and L must be positive.
 4. This is its declaration: **function** Init (I, L : Positive) **return** My_Array;.
 5. You must initialize the elements of the array in the same manner as for the Init procedure described above.
4. Procedure Double doubles each element of an array.
5. Function Diff_Prev_Elem returns — for each element of an input array A — an array with the difference between an element of array A and the previous element.
 1. For the first element, the difference must be zero.
 2. For example:
 - **INPUT:** (2, 5, 15)
 - **RETURN** of Diff_Prev_Elem: (0, 3, 10), where
 - 0 is the constant difference for the first element;
 - 5 - 2 = 3 is the difference between the second and the first elements of the input array;
 - 15 - 5 = 10 is the difference between the third and the second elements of the input array.

Remarks:

1. For an array A, you can retrieve the index of the last element with the attribute '**Last**'.
 1. For example: Y : **Positive** := A'Last;
 2. This can be useful during the implementation of procedure Init.
2. For the implementation of the Init function, you can call the Init procedure to initialize the elements. By doing this, you avoid code duplication.
3. Some hints about attributes:
 1. You can use the range attribute (A'Range) to retrieve the range of an array A.
 2. You can also use the range attribute in the declaration of another array (e.g.: B : My_Array (A'Range)).
 3. Alternatively, you can use the A'First and A'Last attributes in an array declaration.

Listing 7: unconstrained_arrays.ads

```
1 package Unconstrained_Arrays is
2
3     -- Complete the type and subprogram declarations:
4     --
5     -- type My_Array is ...;
6     --
7     -- procedure Init ...;
8
9     function Init (I, L : Positive) return My_Array;
10
11    -- procedure Double ...;
12    --
13    -- function Diff_Prev_Elem ...;
14
15 end Unconstrained_Arrays;
```

Listing 8: unconstrained_arrays.adb

```
1 package body Unconstrained_Arrays is
2
3     -- Implement the subprograms:
4     --
5
6     -- procedure Init is...
7
8     -- function Init (L : Positive) return My_Array is...
9
10    -- procedure Double ... is...
11
12    -- function Diff_Prev_Elem ... is...
13
14 end Unconstrained_Arrays;
```

Listing 9: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Unconstrained_Arrays; use Unconstrained_Arrays;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Init_Chk,
9          Init_Proc_Chk,
10         Double_Chk,
11         Diff_Prev_Chk,
12         Diff_Prev_Single_Chk);
13
14 procedure Check (TC : Test_Case_Index) is
15     AA : My_Array (1 .. 5);
16     AB : My_Array (5 .. 9);
17
18     procedure Display (A : My_Array) is
19     begin
20         for I in A'Range loop
21             Put_Line (Integer'Image (A (I)));
22         end loop;
23     end Display;
24
25     procedure Local_Init (A : in out My_Array) is
```

(continues on next page)

(continued from previous page)

```

26 begin
27   A := (1, 2, 5, 10, -10);
28 end Local_Init;
29
30 begin
31   case TC is
32     when Init_Chk =>
33       AA := Init (AA'First, AA'Length);
34       AB := Init (AB'First, AB'Length);
35       Display (AA);
36       Display (AB);
37     when Init_Proc_Chk =>
38       Init (AA);
39       Init (AB);
40       Display (AA);
41       Display (AB);
42     when Double_Chk =>
43       Local_Init (AB);
44       Double (AB);
45       Display (AB);
46     when Diff_Prev_Chk =>
47       Local_Init (AB);
48       AB := Diff_Prev_Elem (AB);
49       Display (AB);
50     when Diff_Prev_Single_Chk =>
51       declare
52         A1 : My_Array (1 .. 1) := (1 => 42);
53       begin
54         A1 := Diff_Prev_Elem (A1);
55         Display (A1);
56       end;
57     end case;
58 end Check;
59
60 begin
61   if Argument_Count < 1 then
62     Put_Line ("ERROR: missing arguments! Exiting...");
63     return;
64   elsif Argument_Count > 1 then
65     Put_Line ("Ignoring additional arguments...");
66   end if;
67
68   Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

92.4 Product info

Goal: create a system to keep track of quantities and prices of products.

Steps:

1. Implement the Product_Info_Pkg package.
 1. Declare the array type Product_Infos.
 2. Declare the array type Currency_Array.
 3. Implement the Total procedure.
 4. Implement the Total function returning an array of Currency_Array type.

5. Implement the Total function returning a single value of Currency type.

Requirements:

1. Quantity of an individual product is represented by the Quantity subtype.
2. Price of an individual product is represented by the Currency subtype.
3. Record type Product_Info deals with information for various products.
4. Array type Product_Infos is used to represent a list of products.
5. Array type Currency_Array is used to represent a list of total values of individual products (see more details below).
6. Procedure Total receives an input array of products.
 1. It outputs an array with the total value of each product using the Currency_Array type.
 2. The total value of an individual product is calculated by multiplying the quantity for this product by its price.
7. Function Total returns an array of Currency_Array type.
 1. This function has the same purpose as the procedure Total.
 2. The difference is that the function returns an array instead of providing this array as an output parameter.
8. The second function Total returns a single value of Currency type.
 1. This function receives an array of products.
 2. It returns a single value corresponding to the total value for all products in the system.

Remarks:

1. You can use Currency (Q) to convert from an element Q of Quantity type to the Currency type.
 1. As you might remember, Ada requires an explicit conversion in calculations where variables of both integer and floating-point types are used.
 2. In our case, the Quantity subtype is based on the **Integer** type and the Currency subtype is based on the **Float** type, so a conversion is necessary in calculations using those types.

Listing 10: product_info_pkg.ads

```
1 package Product_Info_Pkg is
2
3     subtype Quantity is Natural;
4
5     subtype Currency is Float;
6
7     type Product_Info is record
8         Units : Quantity;
9         Price : Currency;
10    end record;
11
12    -- Complete the type declarations:
13    --
14    -- type Product_Infos is ...
15    --
16    -- type Currency_Array is ...
17
18    procedure Total (P    : Product_Infos;
```

(continues on next page)

(continued from previous page)

```

19          Tot : out Currency_Array);
20
21      function Total (P : Product_Infos) return Currency_Array;
22
23      function Total (P : Product_Infos) return Currency;
24
25 end Product_Info_Pkg;
```

Listing 11: product_info_pkg.adb

```

1 package body Product_Info_Pkg is
2
3     -- Complete the subprogram implementations:
4     --
5
6     -- procedure Total (P   : Product_Infos;
7     --                     Tot : out Currency_Array) is ...
8
9     -- function Total (P : Product_Infos) return Currency_Array is ...
10
11    -- function Total (P : Product_Infos) return Currency is ...
12
13 end Product_Info_Pkg;
```

Listing 12: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Product_Info_Pkg;     use Product_Info_Pkg;
5
6 procedure Main is
7     package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
8
9     type Test_Case_Index is
10        (Total_Func_Chk,
11         Total_Proc_Chk,
12         Total_Value_Chk);
13
14     procedure Check (TC : Test_Case_Index) is
15         subtype Test_Range is Positive range 1 .. 5;
16
17         P      : Product_Infos (Test_Range);
18         Tots : Currency_Array (Test_Range);
19         Tot   : Currency;
20
21     procedure Display (Tots : Currency_Array) is
22     begin
23         for I in Tots'Range loop
24             Currency_IO.Put (Tots (I));
25             New_Line;
26         end loop;
27     end Display;
28
29     procedure Local_Init (P : in out Product_Infos) is
30     begin
31         P := ((1,    0.5),
32                (2,   10.0),
33                (5,   40.0),
34                (10,  10.0),
```

(continues on next page)

(continued from previous page)

```

35         (10, 20.0));
36 end Local_Init;
37
38 begin
39     Currency_I0.Default_Fore := 1;
40     Currency_I0.Default_Aft  := 2;
41     Currency_I0.Default_Exp  := 0;
42
43     case TC is
44     when Total_Func_Chk =>
45         Local_Init (P);
46         Tots := Total (P);
47         Display (Tots);
48     when Total_Proc_Chk =>
49         Local_Init (P);
50         Total (P, Tots);
51         Display (Tots);
52     when Total_Value_Chk =>
53         Local_Init (P);
54         Tot := Total (P);
55         Currency_I0.Put (Tot);
56         New_Line;
57     end case;
58 end Check;
59
60 begin
61     if Argument_Count < 1 then
62         Put_Line ("ERROR: missing arguments! Exiting... ");
63         return;
64     elsif Argument_Count > 1 then
65         Put_Line ("Ignoring additional arguments... ");
66     end if;
67
68     Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

92.5 String_10

Goal: work with constrained string types.

Steps:

1. Implement the Strings_10 package.
 1. Declare the String_10 type.
 2. Implement the To_String_10 function.

Requirements:

1. The constrained string type String_10 is an array of ten characters.
2. Function To_String_10 returns constrained strings of String_10 type based on an input parameter of **String** type.
 - For strings that are more than 10 characters, omit everything after the 11th character.
 - For strings that are fewer than 10 characters, pad the string with '' characters until it is 10 characters.

Remarks:

1. Declaring `String_10` as a subtype of `String` is the easiest way.
 - You may declare it as a new type as well. However, this requires some adaptations in the Main test procedure.
2. You can use `Integer'Min` to calculate the minimum of two integer values.

Listing 13: strings_10.ads

```

1 package Strings_10 is
2
3   -- Complete the type and subprogram declarations:
4   --
5
6   -- subtype String_10 is ...;
7
8   -- Using "type String_10 is..." is possible, too. However, it
9   -- requires a custom Put_Line procedure that is called in Main:
10  -- procedure Put_Line (S : String_10);
11
12  -- function To_String_10 ...;
13
14 end Strings_10;

```

Listing 14: strings_10.adb

```

1 package body Strings_10 is
2
3   -- Complete the subprogram declaration and implementation:
4   --
5   -- function To_String_10 ... is
6
7 end Strings_10;

```

Listing 15: main.adb

```

1 with Ada.Command_Line;    use Ada.Command_Line;
2 with Ada.Text_IO;         use Ada.Text_IO;
3
4 with Strings_10;          use Strings_10;
5
6 procedure Main is
7   type Test_Case_Index is
8     (String_10_Long_Chk,
9      String_10_Short_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12    SL   : constant String := "And this is a long string just for testing...";
13    SS   : constant String := "Hey!";
14    S_10 : String_10;
15
16 begin
17   case TC is
18     when String_10_Long_Chk =>
19       S_10 := To_String_10 (SL);
20       Put_Line (String (S_10));
21     when String_10_Short_Chk =>
22       S_10 := (others => ' ');
23       S_10 := To_String_10 (SS);
24       Put_Line (String (S_10));
25   end case;
26 end Check;

```

(continues on next page)

(continued from previous page)

```
27 begin
28   if Argument_Count < 1 then
29     Ada.Text_Io.Put_Line ("ERROR: missing arguments! Exiting... ");
30     return;
31   elsif Argument_Count > 1 then
32     Ada.Text_Io.Put_Line ("Ignoring additional arguments... ");
33   end if;
34
35   Check (Test_Case_Index'Value (Argument (1)));
36 end Main;
```

92.6 List of Names

Goal: create a system for a list of names and ages.

Steps:

1. Implement the Names_Ages package.
 1. Declare the People_Array array type.
 2. Complete the declaration of the People record type with the People_A element of People_Array type.
 3. Implement the Add procedure.
 4. Implement the Reset procedure.
 5. Implement the Get function.
 6. Implement the Update procedure.
 7. Implement the Display procedure.

Requirements:

1. Each person is represented by the Person type, which is a record containing the name and the age of that person.
2. People_Array is an unconstrained array of Person type with a positive range.
3. The Max_People constant is set to 10.
4. Record type People contains:
 1. The People_A element of People_Array type.
 2. This array must be constrained by the Max_People constant.
5. Procedure Add adds a person to the list.
 1. By default, the age of this person is set to zero in this procedure.
6. Procedure Reset resets the list.
7. Function Get retrieves the age of a person from the list.
8. Procedure Update updates the age of a person in the list.
9. Procedure Display shows the complete list using the following format:
 1. The first line must be LIST OF NAMES:. It is followed by the name and age of each person in the next lines.
 2. For each person on the list, the procedure must display the information in the following format:

```
NAME: XXXX
AGE: YY
```

Remarks:

1. In the implementation of procedure Add, you may use an index to indicate the last valid position in the array — see Last_Valid in the code below.
2. In the implementation of procedure Display, you should use the Trim function from the Ada.Strings.Fixed package to format the person's name — for example: Trim (P.Name, Right).
3. You may need the **Integer'Min** (A, B) and the **Integer'Max** (A, B) functions to get the minimum and maximum values in a comparison between two integer values A and B.
4. Fixed-length strings can be initialized with whitespaces using the **others** syntax. For example: S : String_10 := (**others** => ' ');
5. You may implement additional subprograms to deal with other types declared in the Names_Ages package below, such as the Name_Type and the Person type.
 1. For example, a function To_Name_Type to convert from **String** to Name_Type might be useful.
 2. Take a moment to reflect on which additional subprograms could be useful as well.

Listing 16: names_ages.ads

```

1  package Names_Ages is
2
3    Max_People : constant Positive := 10;
4
5    subtype Name_Type is String (1 .. 50);
6
7    type Age_Type is new Natural;
8
9    type Person is record
10      Name : Name_Type;
11      Age  : Age_Type;
12    end record;
13
14    -- Add type declaration for People_Array record:
15    --
16    -- type People_Array is ...;
17
18    -- Replace type declaration for People record. You may use the
19    -- following template:
20    --
21    -- type People is record
22    --   People_A : People_Array ...;
23    --   Last_Valid : Natural;
24    -- end record;
25    --
26    type People is null record;
27
28  procedure Reset (P : in out People);
29
30  procedure Add (P    : in out People;
31                 Name : String);
32
33  function Get (P    : People;
34                Name : String) return Age_Type;
35
```

(continues on next page)

(continued from previous page)

```

36  procedure Update (P    : in out People;
37          Name  : String;
38          Age   : Age_Type);
39
40  procedure Display (P : People);
41
42 end Names_Ages;

```

Listing 17: names_ages.adb

```

1  with Ada.Text_Io;      use Ada.Text_Io;
2  with Ada.Strings;       use Ada.Strings;
3  with Ada.Strings.Fixed; use Ada.Strings.Fixed;
4
5 package body Names_Ages is
6
7  procedure Reset (P : in out People) is
8  begin
9    null;
10 end Reset;
11
12 procedure Add (P    : in out People;
13                 Name : String) is
14 begin
15   null;
16 end Add;
17
18 function Get (P    : People;
19                Name : String) return Age_Type is
20 begin
21   return 0;
22 end Get;
23
24 procedure Update (P    : in out People;
25                     Name : String;
26                     Age  : Age_Type) is
27 begin
28   null;
29 end Update;
30
31 procedure Display (P : People) is
32 begin
33   null;
34 end Display;
35
36 end Names_Ages;

```

Listing 18: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_Io;           use Ada.Text_Io;
3
4  with Names_Ages;           use Names_Ages;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Names_Ages_Chk,
9       Get_Age_Chk);
10
11 procedure Check (TC : Test_Case_Index) is

```

(continues on next page)

(continued from previous page)

```

12      P : People;
13
14  begin
15    case TC is
16      when Names_Ages_Chk =>
17        Reset (P);
18        Add (P, "John");
19        Add (P, "Patricia");
20        Add (P, "Josh");
21        Display (P);
22        Update (P, "John", 18);
23        Update (P, "Patricia", 35);
24        Update (P, "Josh", 53);
25        Display (P);
26      when Get_Age_Chk =>
27        Reset (P);
28        Add (P, "Peter");
29        Update (P, "Peter", 45);
30        Put_Line ("Peter is "
31                   & Age_Type'Image (Get (P, "Peter"))
32                   & " years old.");
33    end case;
34  end Check;

35 begin
36  if Argument_Count < 1 then
37    Ada.Text_Io.Put_Line ("ERROR: missing arguments! Exiting...");
38    return;
39  elsif Argument_Count > 1 then
40    Ada.Text_Io.Put_Line ("Ignoring additional arguments...");
41  end if;
42
43  Check (Test_Case_Index'Value (Argument (1)));
44 end Main;

```


MORE ABOUT TYPES

93.1 Aggregate Initialization

Goal: initialize records and arrays using aggregates.

Steps:

1. Implement the Aggregates package.
 1. Create the record type Rec.
 2. Create the array type Int_Arr.
 3. Implement the Init procedure that outputs a record of Rec type.
 4. Implement the Init_Some procedure.
 5. Implement the Init procedure that outputs an array of Int_Arr type.

Requirements:

1. Record type Rec has four components of **Integer** type. These are the components with the corresponding default values:
 - W = 10
 - X = 11
 - Y = 12
 - Z = 13
2. Array type Int_Arr has 20 elements of **Integer** type (with indices ranging from 1 to 20).
3. The first Init procedure outputs a record of Rec type where:
 1. X is initialized with 100,
 2. Y is initialized with 200, and
 3. the remaining elements use their default values.
4. Procedure Init_Some outputs an array of Int_Arr type where:
 1. the first five elements are initialized with the value 99, and
 2. the remaining elements are initialized with the value 100.
5. The second Init procedure outputs an array of Int_Arr type where:
 1. all elements are initialized with the value 5.

Listing 1: aggregates.ads

```

1 package Aggregates is
2
3     -- type Rec is ...;
4
5     -- type Int_Arr is ...;
6
7     procedure Init;
8
9     -- procedure Init_Some ...;
10
11    -- procedure Init ...;
12
13 end Aggregates;
```

Listing 2: aggregates.adb

```

1 package body Aggregates is
2
3     procedure Init is null;
4
5 end Aggregates;
```

Listing 3: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Aggregates;       use Aggregates;
5
6 procedure Main is
7     -- Remark: the following line is not relevant.
8     F : array (1 .. 10) of Float := (others => 42.42)
9     with Unreferenced;
10
11    type Test_Case_Index is
12        (Default_Record_Chk,
13         Init_Record_Chk,
14         Init_Some_Array_Chk,
15         Init_Array_Chk);
16
17    procedure Check (TC : Test_Case_Index) is
18        A : Int_Arr;
19        R : Rec;
20        DR : constant Rec := (others => <>);
21
22    begin
23        case TC is
24            when Default_Record_Chk =>
25                R := DR;
26                Put_Line ("Record Default:");
27                Put_Line ("W => " & Integer'Image (R.W));
28                Put_Line ("X => " & Integer'Image (R.X));
29                Put_Line ("Y => " & Integer'Image (R.Y));
30                Put_Line ("Z => " & Integer'Image (R.Z));
31            when Init_Record_Chk =>
32                Init (R);
33                Put_Line ("Record Init:");
34                Put_Line ("W => " & Integer'Image (R.W));
35                Put_Line ("X => " & Integer'Image (R.X));
                Put_Line ("Y => " & Integer'Image (R.Y));
```

(continues on next page)

(continued from previous page)

```

36 Put_Line ("Z => " & Integer'Image (R.Z));
37 when Init_Some_Arr_Chk =>
38   Init_Some (A);
39   Put_Line ("Array Init_Some:");
40   for I in A'Range loop
41     Put_Line (Integer'Image (I) & " "
42               & Integer'Image (A (I)));
43   end loop;
44 when Init_Arr_Chk =>
45   Init (A);
46   Put_Line ("Array Init:");
47   for I in A'Range loop
48     Put_Line (Integer'Image (I) & " "
49               & Integer'Image (A (I)));
50   end loop;
51 end case;
52 end Check;

53
54 begin
55   if Argument_Count < 1 then
56     Put_Line ("ERROR: missing arguments! Exiting...");
57     return;
58   elsif Argument_Count > 1 then
59     Put_Line ("Ignoring additional arguments...");
60   end if;
61
62   Check (Test_Case_Index'Value (Argument (1)));
63 end Main;

```

93.2 Versioning

Goal: implement a simple package for source-code versioning.

Steps:

1. Implement the Versioning package.
 1. Declare the record type Version.
 2. Implement the Convert function that returns a string.
 3. Implement the Convert function that returns a floating-point number.

Requirements:

1. Record type Version has the following components of **Natural** type:
 1. Major,
 2. Minor, and
 3. Maintenance.
2. The first Convert function returns a string containing the version number.
3. The second Convert function returns a floating-point value.
 1. For this floating-point value:
 1. the number before the decimal point must correspond to the major number, and
 2. the number after the decimal point must correspond to the minor number.

3. the maintenance number is ignored.
2. For example, version "1.3.5" is converted to the floating-point value 1.3.
3. An obvious limitation of this function is that it can only handle one-digit numbers for the minor component.
 - For example, we cannot convert version "1.10.0" to a reasonable value with the approach described above. The result of the call `Convert ((1, 10, 0))` is therefore unspecified.
 - For the scope of this exercise, only version numbers with one-digit components are checked.

Remarks:

1. We use overloading for the `Convert` functions.
2. For the function `Convert` that returns a string, you can make use of the `Image_Trim` function, as indicated in the source-code below — see package body of `Versioning`.

Listing 4: `versioning.ads`

```
1 package Versioning is
2
3     -- type Version is record...
4
5     -- function Convert ...
6
7     -- function Convert
8
9 end Versioning;
```

Listing 5: `versioning.adb`

```
1 with Ada.Strings; use Ada.Strings;
2 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3
4 package body Versioning is
5
6     function Image_Trim (N : Natural) return String is
7         S_N : constant String := Trim (Natural'Image (N), Left);
8     begin
9         return S_N;
10    end Image_Trim;
11
12    -- function Convert ...
13    --   S_Major : constant String := Image_Trim (V.Major);
14    --   S_Minor : constant String := Image_Trim (V.Minor);
15    --   S_Maint : constant String := Image_Trim (V.Maintenance);
16    -- begin
17    -- end Convert;
18
19    -- function Convert ...
20    -- begin
21    -- end Convert;
22
23 end Versioning;
```

Listing 6: `main.adb`

```
1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
```

(continues on next page)

(continued from previous page)

```

4  with Versioning;          use Versioning;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Ver_String_Chk,
9       Ver_Float_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12    V : constant Version := (1, 3, 23);
13  begin
14    case TC is
15      when Ver_String_Chk =>
16          Put_Line (Convert (V));
17      when Ver_Float_Chk =>
18          Put_Line (Float'Image (Convert (V)));
19    end case;
20  end Check;
21
22 begin
23  if Argument_Count < 1 then
24    Put_Line ("ERROR: missing arguments! Exiting...");  

25    return;
26  elsif Argument_Count > 1 then
27    Put_Line ("Ignoring additional arguments...");  

28  end if;
29
30  Check (Test_Case_Index'Value (Argument (1)));
31 end Main;

```

93.3 Simple todo list

Goal: implement a simple to-do list system.

Steps:

1. Implement the Todo_Lists package.
 1. Declare the Todo_Item type.
 2. Declare the Todo_List type.
 3. Implement the Add procedure.
 4. Implement the Display procedure.

Requirements:

1. Todo_Item type is used to store a to-do item.
 1. It should be implemented as an access type to strings.
2. Todo_Items type is an array of to-do items.
 1. It should be implemented as an unconstrained array with positive range.
3. Todo_List type is the container for all to-do items.
 1. This record type must have a discriminant for the maximum number of elements of the list.
 2. In order to store the to-do items, it must contain a component named Items of Todo_Items type.
 3. Don't forget to keep track of the last element added to the list!

- You should declare a Last component in the record.
4. Procedure Add adds items (of Todo_Item type) to the list (of Todo_List type).
 1. This requires allocating a string for the access type.
 2. An item can only be added to the list if the list isn't full yet — see next point for details on error handling.
 5. Since the number of items that can be stored on the list is limited, the list might eventually become full in a call to Add.
 1. You must write code in the implementation of the Add procedure that verifies this condition.
 2. If the procedure detects that the list is full, it must display the following message: "ERROR: list is full!".
 6. Procedure Display is used to display all to-do items.
 1. The header (first line) must be T0-DO LIST.
 2. It must display one item per line.

Remarks:

1. We use access types and unconstrained arrays in the implementation of the Todo_Lists package.

Listing 7: todo_lists.ads

```
1 package Todo_Lists is
2
3     -- Replace by actual type declaration
4     type Todo_Item is null record;
5
6     -- Replace by actual type declaration
7     type Todo_Items is null record;
8
9     -- Replace by actual type declaration
10    type Todo_List is null record;
11
12    procedure Add (Todos : in out Todo_List;
13                  Item   : String);
14
15    procedure Display (Todos : Todo_List);
16
17 end Todo_Lists;
```

Listing 8: todo_lists.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Todo_Lists is
4
5     procedure Add (Todos : in out Todo_List;
6                     Item   : String) is
7     begin
8         Put_Line ("ERROR: list is full!");
9     end Add;
10
11    procedure Display (Todos : Todo_List) is
12    begin
13        null;
14    end Display;
```

(continues on next page)

(continued from previous page)

```
15
16 end Todo_Lists;
```

Listing 9: main.adb

```
1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Todo_Lists;         use Todo_Lists;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Todo_List_Chk);
9
10   procedure Check (TC : Test_Case_Index) is
11     T : Todo_List (10);
12   begin
13     case TC is
14       when Todo_List_Chk =>
15         Add (T, "Buy milk");
16         Add (T, "Buy tea");
17         Add (T, "Buy present");
18         Add (T, "Buy tickets");
19         Add (T, "Pay electricity bill");
20         Add (T, "Schedule dentist appointment");
21         Add (T, "Call sister");
22         Add (T, "Revise spreadsheet");
23         Add (T, "Edit entry page");
24         Add (T, "Select new design");
25         Add (T, "Create upgrade plan");
26         Display (T);
27     end case;
28   end Check;
29
30 begin
31   if Argument_Count < 1 then
32     Put_Line ("ERROR: missing arguments! Exiting... ");
33     return;
34   elsif Argument_Count > 1 then
35     Put_Line ("Ignoring additional arguments... ");
36   end if;
37
38   Check (Test_Case_Index'Value (Argument (1)));
39 end Main;
```

93.4 Price list

Goal: implement a list containing prices

Steps:

1. Implement the Price_Lists package.
 1. Declare the Price_Type type.
 2. Declare the Price_List record.
 3. Implement the Reset procedure.
 4. Implement the Add procedure.

5. Implement the Get function.
6. Implement the Display procedure.

Requirements:

1. Price_Type is a decimal fixed-point data type with a delta of two digits (e.g. 0.01) and twelve digits in total.
2. Price_List is a record type that contains the price list.
 1. This record type must have a discriminant for the maximum number of elements of the list.
3. Procedure Reset resets the list.
4. Procedure Add adds a price to the list.
 1. You should keep track of the last element added to the list.
5. Function Get retrieves a price from the list using an index.
 1. This function returns a record instance of Price_Result type.
 2. Price_Result is a variant record containing:
 1. the Boolean component Ok, and
 2. the component Price (of Price_Type).
3. The returned value of Price_Result type is one of the following:
 1. If the index specified in a call to Get contains a valid (initialized) price, then
 - Ok is set to **True**, and
 - the Price component contains the price for that index.
 2. Otherwise:
 - Ok is set to **False**, and
 - the Price component is not available.
6. Procedure Display shows all prices from the list.
 1. The header (first line) must be PRICE LIST.
 2. The remaining lines contain one price per line.
 3. For example:
 - For the following code:

```
procedure Test is
    L : Price_List (10);
begin
    Reset (L);
    Add (L, 1.45);
    Add (L, 2.37);
    Display (L);
end Test;
```

- The output is:

```
PRICE LIST
1.45
2.37
```

Remarks:

1. To implement the package, you'll use the following features of the Ada language:

1. decimal fixed-point types;
 2. records with discriminants;
 3. dynamically-sized record types;
 4. variant records.
2. For record type `Price_List`, you may use an unconstrained array as a component of the record and use the discriminant in the component declaration.

Listing 10: price_lists.ads

```

1 package Price_Lists is
2
3   -- Replace by actual type declaration
4   type Price_Type is new Float;
5
6   -- Replace by actual type declaration
7   type Price_List is null record;
8
9   -- Replace by actual type declaration
10  type Price_Result is null record;
11
12 procedure Reset (Prices : in out Price_List);
13
14 procedure Add (Prices : in out Price_List;
15                 Item   : Price_Type);
16
17 function Get (Prices : Price_List;
18                Idx    : Positive) return Price_Result;
19
20 procedure Display (Prices : Price_List);
21
22 end Price_Lists;

```

Listing 11: price_lists.adb

```

1 package body Price_Lists is
2
3   procedure Reset (Prices : in out Price_List) is
4   begin
5     null;
6   end Reset;
7
8   procedure Add (Prices : in out Price_List;
9                 Item   : Price_Type) is
10  begin
11    null;
12  end Add;
13
14  function Get (Prices : Price_List;
15                Idx    : Positive) return Price_Result is
16  begin
17    null;
18  end Get;
19
20  procedure Display (Prices : Price_List) is
21  begin
22    null;
23  end Display;
24
25 end Price_Lists;

```

Listing 12: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Price_Lists;       use Price_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Price_Type_Chk,
9           Price_List_Chk,
10          Price_List_Get_Chk);
11
12  procedure Check (TC : Test_Case_Index) is
13      L : Price_List (10);
14
15  procedure Local_Init_List is
16  begin
17      Reset (L);
18      Add (L, 1.45);
19      Add (L, 2.37);
20      Add (L, 3.21);
21      Add (L, 4.14);
22      Add (L, 5.22);
23      Add (L, 6.69);
24      Add (L, 7.77);
25      Add (L, 8.14);
26      Add (L, 9.99);
27      Add (L, 10.01);
28  end Local_Init_List;
29
30  procedure Get_Display (Idx : Positive) is
31      R : constant Price_Result := Get (L, Idx);
32  begin
33      Put_Line ("Attempt Get # " & Positive'Image (Idx));
34      if R.Ok then
35          Put_Line ("Element # " & Positive'Image (Idx)
36                      & " => " & Price_Type'Image (R.Price));
37      else
38          declare
39          begin
40              Put_Line ("Element # " & Positive'Image (Idx)
41                          & " => " & Price_Type'Image (R.Price));
42          exception
43              when others =>
44                  Put_Line ("Element not available (as expected)");
45          end;
46      end if;
47
48  end Get_Display;
49
50 begin
51     case TC is
52         when Price_Type_Chk =>
53             Put_Line ("The delta value of Price_Type is "
54                         & Price_Type'Image (Price_Type'Delta) & ";");
55             Put_Line ("The minimum value of Price_Type is "
56                         & Price_Type'Image (Price_Type'First) & ";");
57             Put_Line ("The maximum value of Price_Type is "
58                         & Price_Type'Image (Price_Type'Last) & ";");
59         when Price_List_Chk =>

```

(continues on next page)

(continued from previous page)

```
60      Local_Init_List;
61      Display (L);
62      when Price_List_Get_Chk =>
63          Local_Init_List;
64          Get_Display (5);
65          Get_Display (40);
66      end case;
67  end Check;

68
69 begin
70  if Argument_Count < 1 then
71      Put_Line ("ERROR: missing arguments! Exiting...");
72      return;
73  elsif Argument_Count > 1 then
74      Put_Line ("Ignoring additional arguments...");
75  end if;
76
77  Check (Test_Case_Index'Value (Argument (1)));
78 end Main;
```


94.1 Directions

Goal: create a package that handles directions and geometric angles using a previous implementation.

Steps:

1. Fix the implementation of the Test_Directions procedure.

Requirements:

1. The implementation of the Test_Directions procedure must compile correctly.

Remarks:

1. This exercise is based on the *Directions* exercise from the *Records* (page 997) labs.
 1. In this version, however, Ext_Angle is a private type.
2. In the implementation of the Test_Directions procedure below, the Ada developer tried to initialize All_Directions — an array of Ext_Angle type — with aggregates.
 1. Since we now have a private type, the compiler complains about this initialization.
3. To fix the implementation of the Test_Directions procedure, you should use the appropriate function from the Directions package.
4. The initialization of All_Directions in the code below contains a consistency error where the angle doesn't match the assessed direction.
 1. See if you can spot this error!
 2. This kind of errors can happen when record components that have correlated information are initialized individually without consistency checks — using private types helps to avoid the problem by requiring initialization routines that can enforce consistency.

Listing 1: directions.ads

```
1 package Directions is
2
3     type Angle_Mod is mod 360;
4
5     type Direction is
6         (North,
7          Northwest,
8          West,
9          Southwest,
10         South,
11         Southeast,
12         East);
```

(continues on next page)

(continued from previous page)

```

13  function To_Direction (N : Angle_Mod) return Direction;
14
15  type Ext_Angle is private;
16
17  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
18
19  procedure Display (N : Ext_Angle);
20
21
22  private
23
24  type Ext_Angle is record
25      Angle_Elem    : Angle_Mod;
26      Direction_Elem : Direction;
27  end record;
28
29 end Directions;
```

Listing 2: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          Put_Line ("Angle: "
8                  & Angle_Mod'Image (N.Angle_Elem)
9                  & " => "
10                 & Direction'Image (N.Direction_Elem)
11                 & ".");
12      end Display;
13
14      function To_Direction (N : Angle_Mod) return Direction is
15      begin
16          case N is
17              when 0          => return East;
18              when 1 .. 89    => return Northwest;
19              when 90         => return North;
20              when 91 .. 179  => return Northwest;
21              when 180        => return West;
22              when 181 .. 269 => return Southwest;
23              when 270        => return South;
24              when 271 .. 359 => return Southeast;
25          end case;
26      end To_Direction;
27
28      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29      begin
30          return (Angle_Elem    => N,
31                  Direction_Elem => To_Direction (N));
32      end To_Ext_Angle;
33
34 end Directions;
```

Listing 3: test_directions.adb

```

1  with Directions; use Directions;
2
3  procedure Test_Directions is
```

(continues on next page)

(continued from previous page)

```

4   type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;
5
6   All_Directions : constant Ext_Angle_Array (1 .. 6)
7     := ((0, East),
8           (45, Northwest),
9           (90, North),
10          (91, North),
11          (180, West),
12          (270, South));
13
14 begin
15   for I in All_Directions'Range loop
16     Display (All_Directions (I));
17   end loop;
18
19 end Test_Directions;

```

Listing 4: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Test_Directions;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Direction_Chk);
9
10   procedure Check (TC : Test_Case_Index) is
11   begin
12     case TC is
13       when Direction_Chk =>
14         Test_Directions;
15     end case;
16   end Check;
17
18 begin
19   if Argument_Count < 1 then
20     Put_Line ("ERROR: missing arguments! Exiting..."); return;
21   elsif Argument_Count > 1 then
22     Put_Line ("Ignoring additional arguments..."); end if;
23
24   Check (Test_Case_Index'Value (Argument (1)));
25
26 end Main;

```

94.2 Limited Strings

Goal: work with **limited private** types.

Steps:

1. Implement the Limited_Strings package.
 1. Implement the Copy function.
 2. Implement the = operator.

Requirements:

1. For both Copy and =, the two parameters may refer to strings with different lengths. We'll limit the implementation to just take the minimum length:

1. In case of copying the string "Hello World" to a string with 5 characters, the copied string is "Hello":

```
S1 : constant Lim_String := Init ("Hello World");
S2 :           Lim_String := Init (5);
begin
  Copy (From => S1, To => S2);
  Put_Line (S2);      -- This displays "Hello".
```

2. When comparing "Hello World" to "Hello", the = operator indicates that these strings are equivalent:

```
S1 : constant Lim_String := Init ("Hello World");
S2 : constant Lim_String := Init ("Hello");
begin
  if S1 = S2 then
    -- True => This branch gets selected.
```

2. When copying from a short string to a longer string, the remaining characters of the longer string must be initialized with underscores (_). For example:

```
S1 : constant Lim_String := Init ("Hello");
S2 :           Lim_String := Init (10);
begin
  Copy (From => S1, To => S2);
  Put_Line (S2);      -- This displays "Hello_____".
```

Remarks:

1. As we've discussed in the course:

1. Variables of limited types have the following limitations:
 - they cannot be assigned to;
 - they don't have an equality operator (=).

2. We can, however, define our own, custom subprograms to circumvent these limitations:
 - In order to copy instances of a limited type, we can define a custom Copy procedure.
 - In order to compare instances of a limited type, we can define an = operator.

2. You can use the Min_Last constant — which is already declared in the implementation of these subprograms — in the code you write.

3. Some details about the Limited_Strings package:

1. The Lim_String type acts as a container for strings.
 1. In the the private part, Lim_String is declared as an access type to a **String**.
2. There are two versions of the Init function that initializes an object of Lim_String type:
 1. The first one takes another string.
 2. The second one receives the number of characters for a string *container*.
3. Procedure Put_Line displays object of Lim_String type.
4. The design and implementation of the Limited_Strings package is very simplistic.

1. A good design would have better handling of access types, for example.

Listing 5: limited_strings.ads

```

1 package Limited_Strings is
2
3   type Lim_String is limited private;
4
5   function Init (S : String) return Lim_String;
6
7   function Init (Max : Positive) return Lim_String;
8
9   procedure Put_Line (LS : Lim_String);
10
11  procedure Copy (From :          Lim_String;
12                  To    : in out Lim_String);
13
14  function "=" (Ref, Dut : Lim_String) return Boolean;
15
16 private
17
18   type Lim_String is access String;
19
20 end Limited_Strings;

```

Listing 6: limited_strings.adb

```

1 with Ada.Text_IO;
2
3 package body Limited_Strings
4 is
5
6   function Init (S : String) return Lim_String is
7     LS : constant Lim_String := new String'(S);
8   begin
9     return LS;
10  end Init;
11
12  function Init (Max : Positive) return Lim_String is
13    LS : constant Lim_String := new String (1 .. Max);
14  begin
15    LS.all := (others => '_');
16    return LS;
17  end Init;
18
19  procedure Put_Line (LS : Lim_String) is
20  begin
21    Ada.Text_Io.Put_Line (LS.all);
22  end Put_Line;
23
24  function Get_Min_Last (A, B : Lim_String) return Positive is
25  begin
26    return Positive'Min (A'Last, B'Last);
27  end Get_Min_Last;
28
29  procedure Copy (From :          Lim_String;
30                  To    : in out Lim_String) is
31    Min_Last : constant Positive := Get_Min_Last (From, To);
32  begin
33    -- Complete the implementation!
34    null;
35  end;

```

(continues on next page)

(continued from previous page)

```

36
37     function "=" (Ref, Dut : Lim_String) return Boolean is
38         Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
39     begin
40         -- Complete the implementation!
41         return True;
42     end;
43
44 end Limited_Strings;

```

Listing 7: check_lim_string.adb

```

1  with Ada.Text_IO;      use Ada.Text_IO;
2
3  with Limited_Strings; use Limited_Strings;
4
5  procedure Check_Lim_String is
6      S : constant String := "-----";
7      S1 : constant Lim_String := Init ("Hello World");
8      S2 : constant Lim_String := Init (30);
9      S3 : Lim_String := Init (5);
10     S4 : Lim_String := Init (S & S & S);
11 begin
12     Put ("S1 => ");
13     Put_Line (S1);
14     Put ("S2 => ");
15     Put_Line (S2);
16
17     if S1 = S2 then
18         Put_Line ("S1 is equal to S2.");
19     else
20         Put_Line ("S1 isn't equal to S2.");
21     end if;
22
23     Copy (From => S1, To => S3);
24     Put ("S3 => ");
25     Put_Line (S3);
26
27     if S1 = S3 then
28         Put_Line ("S1 is equal to S3.");
29     else
30         Put_Line ("S1 isn't equal to S3.");
31     end if;
32
33     Copy (From => S1, To => S4);
34     Put ("S4 => ");
35     Put_Line (S4);
36
37     if S1 = S4 then
38         Put_Line ("S1 is equal to S4.");
39     else
40         Put_Line ("S1 isn't equal to S4.");
41     end if;
42 end Check_Lim_String;

```

Listing 8: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3

```

(continues on next page)

(continued from previous page)

```

4  with Check_Lim_String;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Lim_String_Chk);
9
10   procedure Check (TC : Test_Case_Index) is
11   begin
12     case TC is
13       when Lim_String_Chk =>
14         Check_Lim_String;
15     end case;
16   end Check;
17
18 begin
19   if Argument_Count < 1 then
20     Put_Line ("ERROR: missing arguments! Exiting... ");
21     return;
22   elsif Argument_Count > 1 then
23     Put_Line ("Ignoring additional arguments... ");
24   end if;
25
26   Check (Test_Case_Index'Value (Argument (1)));
27 end Main;

```

94.3 Bonus exercise

In previous labs, we had many source-code snippets containing records that could be declared private. The source-code for the exercise above (*Directions*) is an example: we've modified the type declaration of Ext_Angle, so that the record is now private. Encapsulating the record components — by declaring record components in the private part — makes the code safer. Also, because many of the code snippets weren't making use of record components directly (but handling record types via the API instead), they continue to work fine after these modifications.

This exercise doesn't contain any source-code. In fact, the **goal** here is to modify previous labs, so that the record declarations are made private. You can look into those labs, modify the type declarations, and recompile the code. The corresponding test-cases must still pass.

If no other changes are needed apart from changes in the declaration, then that indicates we have used good programming techniques in the original code. On the other hand, if further changes are needed, then you should investigate why this is the case.

Also note that, in some cases, you can move support types into the private part of the specification without affecting its compilation. This is the case, for example, for the People_Array type of the *List of Names* lab mentioned below. You should, in fact, keep only relevant types and subprograms in the public part and move all support declarations to the private part of the specification whenever possible.

Below, you find the selected labs that you can work on, including changes that you should make. In case you don't have a working version of the source-code of previous labs, you can look into the corresponding solutions.

94.3.1 Colors

Chapter: *Records* (page 997)

Steps:

1. Change declaration of RGB type to **private**.

Requirements:

1. Implementation must compile correctly and test cases must pass.

94.3.2 List of Names

Chapter: *Arrays* (page 1007)

Steps:

1. Change declaration of Person and People types to **limited private**.
2. Move type declaration of People_Array to private part.

Requirements:

1. Implementation must compile correctly and test cases must pass.

94.3.3 Price List

Chapter: *More About Types* (page 1025)

Steps:

1. Change declaration of Price_List type to **limited private**.

Requirements:

1. Implementation must compile correctly and test cases must pass.

GENERICS

95.1 Display Array

Goal: create a generic procedure that displays the elements of an array.

Steps:

1. Implement the generic procedure `Display_Array`.

Requirements:

1. Generic procedure `Display_Array` displays the elements of an array.

1. It uses the following scheme:

- First, it displays a header.
- Then, it displays the elements of the array.

2. When displaying the elements, it must:

- use one line per element, and
- include the corresponding index of the array.

3. This is the expected format:

```
<HEADER>
<index #1>: <element #1>
<index #2>: <element #2>
...
...
```

4. For example:

- For the following code:

```
procedure Test is
    A : Int_Array (1 .. 2) := (1, 5);
begin
    Display_Int_Array ("Elements of A", A);
end Test;
```

- The output is:

```
Elements of A
1: 1
2: 5
```

2. These are the formal parameters of the procedure:

1. a range type `T_Range` for the array;
2. a formal type `T_Element` for the elements of the array;

- This type must be declared in such a way that it can be mapped to any type in the instantiation — including record types.
3. an array type T_Array using the T_Range and T_Element types;
 4. a function Image that converts a variable of T_Element type to a **String**.

Listing 1: display_array.ads

```

1 generic
2 procedure Display_Array (Header : String;
3                         A      : T_Array);

```

Listing 2: display_array.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2
3 procedure Display_Array (Header : String;
4                         A      : T_Array) is
5 begin
6   null;
7 end Display_Array;

```

Listing 3: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_Io;      use Ada.Text_Io;
3
4 with Display_Array;
5
6 procedure Main is
7   type Test_Case_Index is (Int_Array_Chk,
8                             Point_Array_Chk);
9
10  procedure Test_Int_Array is
11    type Int_Array is array (Positive range <>) of Integer;
12
13    procedure Display_Int_Array is new
14      Display_Array (T_Range => Positive,
15                      T_Element => Integer,
16                      T_Array    => Int_Array,
17                      Image      => Integer'Image);
18
19    A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
20  begin
21    Display_Int_Array ("Integers", A);
22  end Test_Int_Array;
23
24  procedure Test_Point_Array is
25    type Point is record
26      X : Float;
27      Y : Float;
28    end record;
29
30    type Point_Array is array (Natural range <>) of Point;
31
32    function Image (P : Point) return String is
33    begin
34      return "(" & Float'Image (P.X)
35                  & ", " & Float'Image (P.Y) & ")";
36    end Image;
37

```

(continues on next page)

(continued from previous page)

```

38  procedure Display_Point_Array is new
39      Display_Array (T_Range    => Natural,
40                      T_Element  => Point,
41                      T_Array    => Point_Array,
42                      Image      => Image);
43
44      A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
45                                              (5.0, 2.0), (-0.5, 2.0));
46
47 begin
48     Display_Point_Array ("Points", A);
49 end Test_Point_Array;
50
51 procedure Check (TC : Test_Case_Index) is
52 begin
53     case TC is
54         when Int_Array_Chk =>
55             Test_Int_Array;
56         when Point_Array_Chk =>
57             Test_Point_Array;
58     end case;
59 end Check;
60
61 begin
62     if Argument_Count < 1 then
63         Put_Line ("ERROR: missing arguments! Exiting...");  

64         return;
65     elsif Argument_Count > 1 then
66         Put_Line ("Ignoring additional arguments...");  

67     end if;
68
69     Check (Test_Case_Index'Value (Argument (1)));
end Main;

```

95.2 Average of Array of Float

Goal: create a generic function that calculates the average of an array of floating-point elements.

Steps:

1. Declare and implement the generic function Average.

Requirements:

1. Generic function Average calculates the average of an array containing floating-point values of arbitrary precision.
2. Generic function Average must contain the following formal parameters:
 1. a range type T_Range for the array;
 2. a formal type T_Element that can be mapped to floating-point types of arbitrary precision;
 3. an array type T_Array using T_Range and T_Element;

Remarks:

1. You should use the **Float** type for the accumulator.

Listing 4: average.ads

```
1 generic
2 function Average (A : T_Array) return T_Element;
```

Listing 5: average.adb

```
1 function Average (A : T_Array) return T_Element is
2 begin
3     return 0.0;
4 end Average;
```

Listing 6: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Average;
5
6 procedure Main is
7     type Test_Case_Index is (Float_Array_Chk,
8                               Digits_7_Float_Array_Chk);
9
10    procedure Test_Float_Array is
11        type Float_Array is array (Positive range <>) of Float;
12
13        function Average_Float is new
14            Average (T_Range    => Positive,
15                      T_Element => Float,
16                      T_Array    => Float_Array);
17
18        A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
19    begin
20        Put_Line ("Average: " & Float'Image (Average_Float (A)));
21    end Test_Float_Array;
22
23    procedure Test_Digits_7_Float_Array is
24        type Custom_Float is digits 7 range 0.0 .. 1.0;
25
26        type Float_Array is
27            array (Integer range <>) of Custom_Float;
28
29        function Average_Float is new
30            Average (T_Range    => Integer,
31                      T_Element => Custom_Float,
32                      T_Array    => Float_Array);
33
34        A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
35    begin
36        Put_Line ("Average: "
37                  & Custom_Float'Image (Average_Float (A)));
38    end Test_Digits_7_Float_Array;
39
40    procedure Check (TC : Test_Case_Index) is
41    begin
42        case TC is
43            when Float_Array_Chk =>
44                Test_Float_Array;
45            when Digits_7_Float_Array_Chk =>
46                Test_Digits_7_Float_Array;
47        end case;

```

(continues on next page)

(continued from previous page)

```

48  end Check;
49
50 begin
51  if Argument_Count < 1 then
52    Put_Line ("ERROR: missing arguments! Exiting...");  

53    return;
54  elsif Argument_Count > 1 then
55    Put_Line ("Ignoring additional arguments...");  

56  end if;
57
58  Check (Test_Case_Index'Value (Argument (1)));
59 end Main;

```

95.3 Average of Array of Any Type

Goal: create a generic function that calculates the average of an array of elements of any arbitrary type.

Steps:

1. Declare and implement the generic function Average.
2. Implement the test procedure Test_Item.
 1. Declare the F_I0 package.
 2. Implement the Get_Total function for the Item type.
 3. Implement the Get_Price function for the Item type.
 4. Declare the Average_Total function.
 5. Declare the Average_Price function.

Requirements:

1. Generic function Average calculates the average of an array containing elements of any arbitrary type.
2. Generic function Average has the same formal parameters as in the previous exercise, except for:
 1. T_Element, which is now a formal type that can be mapped to any arbitrary type.
 2. To_Float, which is an *additional* formal parameter.
 - To_Float is a function that converts the arbitrary element of T_Element type to the **Float** type.
3. Procedure Test_Item is used to test the generic Average procedure for a record type (Item).
 1. Record type Item contains the Quantity and Price components.
4. The following functions have to implemented to be used for the formal To_Float function parameter:
 1. For the Decimal type, the function is pretty straightforward: it simply returns the floating-point value converted from the decimal type.
 2. For the Item type, two functions must be created to convert to floating-point type:
 1. Get_Total, which returns the multiplication of the quantity and the price components of the Item type;

2. Get_Price, which returns just the price.
5. The generic function Average must be instantiated as follows:
 1. For the Item type, you must:
 1. declare the Average_Total function (as an instance of Average) using the Get_Total for the To_Float parameter;
 2. declare the Average_Price function (as an instance of Average) using the Get_Price for the To_Float parameter.
 6. You must use the Put procedure from Ada.Text_Io.Float_Io.
 1. The generic standard package Ada.Text_Io.Float_Io must be instantiated as F_Io in the test procedures.
 2. This is the specification of the Put procedure, as described in the appendix A.10.9 of the Ada Reference Manual:

```
procedure Put(Item : in Num;
             Fore : in Field := Default_Fore;
             Aft  : in Field := Default_Aft;
             Exp  : in Field := Default_Exp);
```

3. This is the expected format when calling Put from Float_Io:

Function	Fore	Aft	Exp
Test_Item	3	2	0

Remarks:

1. In this exercise, you'll abstract the Average function from the previous exercises a step further.
 1. In this case, the function shall be able to calculate the average of any arbitrary type — including arrays containing elements of record types.
 2. Since record types can be composed by many components of different types, we need to provide a way to indicate which component (or components) of the record will be used when calculating the average of the array.
 3. This problem is solved by specifying a To_Float function as a formal parameter, which converts the arbitrary element of T_Element type to the **Float** type.
 4. In the implementation of the Average function, we use the To_Float function and calculate the average using a floating-point variable.

Listing 7: average.ads

```
1 generic
2 function Average (A : T_Array) return Float;
```

Listing 8: average.adb

```
1 function Average (A : T_Array) return Float is
2 begin
3     null;
4 end Average;
```

Listing 9: test_item.ads

```
1 procedure Test_Item;
```

Listing 10: test_item.adb

```

1  with Ada.Text_Io;      use Ada.Text_Io;
2
3  with Average;
4
5  procedure Test_Item is
6    type Amount is delta 0.01 digits 12;
7
8    type Item is record
9      Quantity : Natural;
10     Price   : Amount;
11   end record;
12
13  type Item_Array is
14    array (Positive range <>) of Item;
15
16  A : constant Item_Array (1 .. 4)
17    := ((Quantity => 5,      Price => 10.00),
18          (Quantity => 80,     Price => 2.50),
19          (Quantity => 40,     Price => 5.00),
20          (Quantity => 20,     Price => 12.50));
21
22 begin
23   Put ("Average per item & quantity: ");
24   F_Io.Put (Average_Total (A));
25   New_Line;
26
27   Put ("Average price:           ");
28   F_Io.Put (Average_Price (A));
29   New_Line;
30 end Test_Item;

```

Listing 11: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_Io;      use Ada.Text_Io;
3
4  with Test_Item;
5
6  procedure Main is
7    type Test_Case_Index is (Item_Array_Chk);
8
9    procedure Check (TC : Test_Case_Index) is
10 begin
11   case TC is
12     when Item_Array_Chk =>
13       Test_Item;
14   end case;
15 end Check;
16
17 begin
18   if Argument_Count < 1 then
19     Put_Line ("ERROR: missing arguments! Exiting... ");
20     return;
21   elsif Argument_Count > 1 then
22     Put_Line ("Ignoring additional arguments... ");
23   end if;
24
25   Check (Test_Case_Index'Value (Argument (1)));
26 end Main;

```

95.4 Generic list

Goal: create a system based on a generic list to add and displays elements.

Steps:

1. Declare and implement the generic package Gen_List.
 1. Implement the Init procedure.
 2. Implement the Add procedure.
 3. Implement the Display procedure.

Requirements:

1. Generic package Gen_List must have the following subprograms:
 1. Procedure Init initializes the list.
 2. Procedure Add adds an item to the list.
 1. This procedure must contain a Status output parameter that is set to **False** when the list was full — i.e. if the procedure failed while trying to add the item;
 3. Procedure Display displays the complete list.
 1. This includes the *name* of the list and its elements — using one line per element.
 2. This is the expected format:

```
<NAME>
<element #1>
<element #2>
...

```

2. Generic package Gen_List has these formal parameters:
 1. an arbitrary formal type Item;
 2. an unconstrained array type Items of Item element with positive range;
 3. the Name parameter containing the name of the list;
 - This must be a formal input object of **String** type.
 - It must be used in the Display procedure.
 4. an actual array List_Array to store the list;
 - This must be a formal **in out** object of Items type.
 5. the variable Last to store the index of the last element;
 - This must be a formal **in out** object of **Natural** type.
 6. a procedure Put for the Item type.
 - This procedure is used in the Display procedure to display individual elements of the list.
3. The test procedure Test_Int is used to test a list of elements of **Integer** type.
4. For both test procedures, you must:
 1. add missing type declarations;
 2. declare and implement a Put procedure for individual elements of the list;
 3. declare instances of the Gen_List package.
 - For the Test_Int procedure, declare the Int_List package.

Remarks:

1. In previous labs, you've been implementing lists for a variety of types.
 - The *List of Names* exercise from the [Arrays](#) (page 1007) labs is an example.
 - In this exercise, you have to abstract those implementations to create the generic `Gen_List` package.

Listing 12: `gen_list.ads`

```

1 generic
2 package Gen_List is
3
4   procedure Init;
5
6   procedure Add (I      : Item;
7                  Status : out Boolean);
8
9   procedure Display;
10
11 end Gen_List;

```

Listing 13: `gen_list.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Gen_List is
4
5   procedure Init is
6   begin
7     null;
8   end Init;
9
10  procedure Add (I      : Item;
11                  Status : out Boolean) is
12  begin
13    null;
14  end Add;
15
16  procedure Display is
17  begin
18    null;
19  end Display;
20
21 end Gen_List;

```

Listing 14: `test_int.ads`

```

1 procedure Test_Int;

```

Listing 15: `test_int.adb`

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_List;
4
5 procedure Test_Int is
6
7   type Integer_Array is array (Positive range <>) of Integer;
8
9   A : Integer_Array (1 .. 3);

```

(continues on next page)

(continued from previous page)

```

10  L : Natural;
11
12  Success : Boolean;
13
14  procedure Display_Add_Success (Success : Boolean) is
15  begin
16      if Success then
17          Put_Line ("Added item successfully!");
18      else
19          Put_Line ("Couldn't add item!");
20      end if;
21
22  end Display_Add_Success;
23
24 begin
25     Int_List.Init;
26
27     Int_List.Add (2, Success);
28     Display_Add_Success (Success);
29
30     Int_List.Add (5, Success);
31     Display_Add_Success (Success);
32
33     Int_List.Add (7, Success);
34     Display_Add_Success (Success);
35
36     Int_List.Add (8, Success);
37     Display_Add_Success (Success);
38
39     Int_List.Display;
40 end Test_Int;

```

Listing 16: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Test_Int;
5
6  procedure Main is
7      type Test_Case_Index is (Int_Chk);
8
9  procedure Check (TC : Test_Case_Index) is
10 begin
11     case TC is
12         when Int_Chk =>
13             Test_Int;
14     end case;
15 end Check;
16
17 begin
18     if Argument_Count < 1 then
19         Put_Line ("ERROR: missing arguments! Exiting...");
20         return;
21     elsif Argument_Count > 1 then
22         Put_Line ("Ignoring additional arguments...");
23     end if;
24
25     Check (Test_Case_Index'Value (Argument (1)));
26 end Main;

```

EXCEPTIONS

96.1 Uninitialized Value

Goal: implement an enumeration to avoid the use of uninitialized values.

Steps:

1. Implement the Options package.
 1. Declare the Option enumeration type.
 2. Declare the Uninitialized_Value exception.
 3. Implement the Image function.

Requirements:

1. Enumeration Option contains:
 1. the Uninitialized value, and
 2. the actual options:
 - Option_1,
 - Option_2,
 - Option_3.
2. Function Image returns a string for the Option type.
 1. In case the argument to Image is Uninitialized, the function must raise the Uninitialized_Value exception.

Remarks:

1. In this exercise, we employ exceptions as a mechanism to avoid the use of uninitialized values for a certain type.

Listing 1: options.ads

```
1 package Options is
2
3     -- Declare the Option enumeration type!
4     type Option is null record;
5
6     function Image (O : Option) return String;
7
8 end Options;
```

Listing 2: options.adb

```
1 package body Options is
2
3     function Image (O : Option) return String is
4 begin
5     return "";
6 end Image;
7
8 end Options;
```

Listing 3: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with Ada.Exceptions;  use Ada.Exceptions;
4
5 with Options;          use Options;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Options_Chk);
10
11    procedure Check (TC : Test_Case_Index) is
12
13        procedure Check (O : Option) is
14        begin
15            Put_Line (Image (O));
16        exception
17            when E : Uninitialized_Value =>
18                Put_Line (Exception_Message (E));
19        end Check;
20
21    begin
22        case TC is
23        when Options_Chk =>
24            for O in Option loop
25                Check (O);
26            end loop;
27        end case;
28    end Check;
29
30    begin
31        if Argument_Count < 1 then
32            Put_Line ("ERROR: missing arguments! Exiting...");
33            return;
34        elsif Argument_Count > 1 then
35            Put_Line ("Ignoring additional arguments...");
36        end if;
37
38        Check (Test_Case_Index'Value (Argument (1)));
39    end Main;
```

96.2 Numerical Exception

Goal: handle numerical exceptions in a test procedure.

Steps:

1. Add exception handling to the Check_Exception procedure.

Requirements:

1. The test procedure Num_Exception_Test from the Tests package below must be used in the implementation of Check_Exception.
2. The Check_Exception procedure must be extended to handle exceptions as follows:
 1. If the exception raised by Num_Exception_Test is Constraint_Error, the procedure must display the message "Constraint_Error detected!" to the user.
 2. Otherwise, it must display the message associated with the exception.

Remarks:

1. You can use the Exception_Message function to retrieve the message associated with an exception.

Listing 4: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;

```

Listing 5: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10         A := Integer'Last;
11         B := Integer'Last;
12         C := A + B;
13       when Test_2 =>
14         raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;

```

Listing 6: check_exception.adb

```

1 with Tests; use Tests;
2

```

(continues on next page)

(continued from previous page)

```
3 procedure Check_Exception (ID : Test_ID) is
4 begin
5     Num_Exception_Test (ID);
6 end Check_Exception;
```

Listing 7: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with Ada.Exceptions;  use Ada.Exceptions;
4
5 with Tests;           use Tests;
6 with Check_Exception;
7
8 procedure Main is
9     type Test_Case_Index is
10        (Exception_1_Chk,
11         Exception_2_Chk);
12
13 procedure Check (TC : Test_Case_Index) is
14
15     procedure Check_Handle_Exception (ID : Test_ID) is
16     begin
17         Check_Exception (ID);
18     exception
19         when Constraint_Error =>
20             Put_Line ("Constraint_Error"
21                       & " (raised by Check_Exception) detected!");
22         when E : others =>
23             Put_Line (Exception_Name (E)
24                       & " (raised by Check_Exception) detected!");
25     end Check_Handle_Exception;
26
27 begin
28     case TC is
29     when Exception_1_Chk =>
30         Check_Handle_Exception (Test_1);
31     when Exception_2_Chk =>
32         Check_Handle_Exception (Test_2);
33     end case;
34 end Check;
35
36 begin
37     if Argument_Count < 1 then
38         Put_Line ("ERROR: missing arguments! Exiting...");
39         return;
40     elsif Argument_Count > 1 then
41         Put_Line ("Ignoring additional arguments...");
42     end if;
43
44     Check (Test_Case_Index'Value (Argument (1)));
45 end Main;
```

96.3 Re-raising Exceptions

Goal: make use of exception re-raising in a test procedure.

Steps:

1. Declare new exception: Another_Exception.
2. Add exception re-raise to the Check_Exception procedure.

Requirements:

1. Exception Another_Exception must be declared in the Tests package.
2. Procedure Check_Exception must be extended to *re-raise* any exception. When an exception is detected, the procedure must:
 1. display a user message (as implemented in the previous exercise), and then
 2. Raise or *re-raise* exception depending on the exception that is being handled:
 1. In case of Constraint_Error exception, *re-raise* the exception.
 2. In all other cases, raise Another_Exception.

Remarks:

1. In this exercise, you should extend the implementation of the Check_Exception procedure from the previous exercise.
 1. Naturally, you can use the code for the Check_Exception procedure from the previous exercise as a starting point.

Listing 8: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;

```

Listing 9: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10        A := Integer'Last;
11        B := Integer'Last;
12        C := A + B;
13       when Test_2 =>
14        raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");

```

(continues on next page)

(continued from previous page)

```
19  
20 end Tests;
```

Listing 10: check_exception.ads

```
1 with Tests; use Tests;  
2  
3 procedure Check_Exception (ID : Test_ID);
```

Listing 11: check_exception.adb

```
1 procedure Check_Exception (ID : Test_ID) is  
2 begin  
3     Num_Exception_Test (ID);  
4 end Check_Exception;
```

Listing 12: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;  
2 with Ada.Text_IO; use Ada.Text_IO;  
3 with Ada.Exceptions; use Ada.Exceptions;  
4  
5 with Tests; use Tests;  
6 with Check_Exception;  
7  
8 procedure Main is  
9     type Test_Case_Index is  
10        (Exception_1_Chk,  
11         Exception_2_Chk);  
12  
13 procedure Check (TC : Test_Case_Index) is  
14  
15     procedure Check_Handle_Exception (ID : Test_ID) is  
16     begin  
17         Check_Exception (ID);  
18     exception  
19         when Constraint_Error =>  
20             Put_Line ("Constraint_Error"  
21                         & " (raised by Check_Exception) detected!");  
22         when E : others =>  
23             Put_Line (Exception_Name (E)  
24                         & " (raised by Check_Exception) detected!");  
25     end Check_Handle_Exception;  
26  
27 begin  
28     case TC is  
29     when Exception_1_Chk =>  
30         Check_Handle_Exception (Test_1);  
31     when Exception_2_Chk =>  
32         Check_Handle_Exception (Test_2);  
33     end case;  
34 end Check;  
35  
36 begin  
37     if Argument_Count < 1 then  
38         Put_Line ("ERROR: missing arguments! Exiting...");  
39         return;  
40     elsif Argument_Count > 1 then  
41         Put_Line ("Ignoring additional arguments...");  
42     end if;
```

(continues on next page)

(continued from previous page)

```
43      Check (Test_Case_Index'Value (Argument (1)));
44
45  end Main;
```


TASKING

97.1 Display Service

Goal: create a simple service that displays messages to the user.

Steps:

1. Implement the `Display_Services` package.
 1. Declare the task type `Display_Service`.
 2. Implement the `Display` entry for strings.
 3. Implement the `Display` entry for integers.

Requirements:

1. Task type `Display_Service` uses the `Display` entry to display messages to the user.
2. There are two versions of the `Display` entry:
 1. One that receives messages as a string parameter.
 2. One that receives messages as an **Integer** parameter.
3. When a message is received via a `Display` entry, it must be displayed immediately to the user.

Listing 1: `display_services.ads`

```
1 package Display_Services is
2
3 end Display_Services;
```

Listing 2: `display_services.adb`

```
1 package body Display_Services is
2
3 end Display_Services;
```

Listing 3: `main.adb`

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Services; use Display_Services;
5
6 procedure Main is
7   type Test_Case_Index is (Display_Service_Chk);
8
9   procedure Check (TC : Test_Case_Index) is
```

(continues on next page)

(continued from previous page)

```
10  Display : Display_Service;
11 begin
12  case TC is
13    when Display_Service_Chk =>
14      Display.Display ("Hello");
15      delay 0.5;
16      Display.Display ("Hello again");
17      delay 0.5;
18      Display.Display (55);
19      delay 0.5;
20  end case;
21 end Check;

22
23 begin
24  if Argument_Count < 1 then
25    Put_Line ("ERROR: missing arguments! Exiting...");  

26    return;
27  elsif Argument_Count > 1 then
28    Put_Line ("Ignoring additional arguments...");  

29  end if;
30
31  Check (Test_Case_Index'Value (Argument (1)));
32 end Main;
```

97.2 Event Manager

Goal: implement a simple event manager.

Steps:

1. Implement the Event_Managers package.
 1. Declare the task type Event_Manager.
 2. Implement the Start entry.
 3. Implement the Event entry.

Requirements:

1. The event manager has a similar behavior as an alarm
 1. The sole purpose of this event manager is to display the event ID at the correct time.
 2. After the event ID is displayed, the task must finish.
2. The event manager (Event_Manager type) must have two entries:
 1. Start, which starts the event manager with an event ID;
 2. Event, which delays the task until a certain time and then displays the event ID as a user message.
3. The format of the user message displayed by the event manager is Event #<event_id>.
 1. You should use Natural'Image to display the ID (as indicated in the body of the Event_Managers package below).

Remarks:

1. In the Start entry, you can use the **Natural** type for the ID.

2. In the Event entry, you should use the Time type from the Ada.Real_Time package for the time parameter.
3. Note that the test application below creates an array of event managers with different delays.

Listing 4: event_managers.ads

```

1 package Event_Managers is
2
3 end Event_Managers;

```

Listing 5: event_managers.adb

```

1 package body Event_Managers is
2
3     -- Don't forget to display the event ID:
4     --
5     -- Put_Line ("Event #" & Natural'Image (Event_ID));
6
7 end Event_Managers;

```

Listing 6: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Event_Managers;   use Event_Managers;
5 with Ada.Real_Time;    use Ada.Real_Time;
6
7 procedure Main is
8     type Test_Case_Index is (Event_Manager_Chk);
9
10    procedure Check (TC : Test_Case_Index) is
11        Ev_Mng : array (1 .. 5) of Event_Manager;
12    begin
13        case TC is
14            when Event_Manager_Chk =>
15                for I in Ev_Mng'Range loop
16                    Ev_Mng (I).Start (I);
17                end loop;
18                Ev_Mng (1).Event (Clock + Seconds (5));
19                Ev_Mng (2).Event (Clock + Seconds (3));
20                Ev_Mng (3).Event (Clock + Seconds (1));
21                Ev_Mng (4).Event (Clock + Seconds (2));
22                Ev_Mng (5).Event (Clock + Seconds (4));
23        end case;
24    end Check;
25
26 begin
27    if Argument_Count < 1 then
28        Put_Line ("ERROR: missing arguments! Exiting... ");
29        return;
30    elsif Argument_Count > 1 then
31        Put_Line ("Ignoring additional arguments... ");
32    end if;
33
34    Check (Test_Case_Index'Value (Argument (1)));
35 end Main;

```

97.3 Generic Protected Queue

Goal: create a queue container using a protected type.

Steps:

1. Implement the generic package Gen_Queues.
 1. Declare the protected type Queue.
 2. Implement the Empty function.
 3. Implement the Full function.
 4. Implement the Push entry.
 5. Implement the Pop entry.

Requirements:

1. These are the formal parameters for the generic package Gen_Queues:
 1. a formal modular type:
 - This modular type should be used by the Queue to declare an array that stores the elements of the queue.
 - The modulus of the modular type must correspond to the maximum number of elements of the queue.
 2. the data type of the elements of the queue.
 - Select a formal parameter that allows you to store elements of any data type in the queue.
2. These are the operations of the Queue type:
 1. Function Empty indicates whether the queue is empty.
 2. Function Full indicates whether the queue is full.
 3. Entry Push stores an element in the queue.
 4. Entry Pop removes an element from the queue and returns the element via output parameter.

Remarks:

1. In this exercise, we create a queue container by declaring and implementing a protected type (Queue) as part of a generic package (Gen_Queues).
2. As a bonus exercise, you can analyze the body of the Queue_Tests package and understand how the Queue type is used there.
 1. In particular, the procedure Concurrent_Test implements two tasks: T_Producer and T_Consumer. They make use of the queue concurrently.

Listing 7: gen_queues.ads

```
1 package Gen_Queues is
2
3 end Gen_Queues;
```

Listing 8: gen_queues.adb

```
1 package body Gen_Queues is
2
3 end Gen_Queues;
```

Listing 9: queue_tests.ads

```

1 package Queue_Tests is
2
3   procedure Simple_Test;
4
5   procedure Concurrent_Test;
6
7 end Queue_Tests;

```

Listing 10: queue_tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_Queues;
4
5 package body Queue_Tests is
6
7   Max : constant := 10;
8   type Queue_Mod is mod Max;
9
10  procedure Simple_Test is
11    package Queues_Float is new Gen_Queues (Queue_Mod, Float);
12
13    Q_F : Queues_Float.Queue;
14    V   : Float;
15
16  begin
17    V := 10.0;
18    while not Q_F.Full loop
19      Q_F.Push (V);
20      V := V + 1.5;
21    end loop;
22
23    while not Q_F.Empty loop
24      Q_F.Pop (V);
25      Put_Line ("Value from queue: " & Float'Image (V));
26    end loop;
27  end Simple_Test;
28
29  procedure Concurrent_Test is
30    package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
31
32    Q_I : Queues_Integer.Queue;
33
34    task T_Producer;
35    task T_Consumer;
36
37    task body T_Producer is
38      V : Integer := 100;
39    begin
40      for I in 1 .. 2 * Max loop
41        Q_I.Push (V);
42        V := V + 1;
43      end loop;
44    end T_Producer;
45
46    task body T_Consumer is
47      V : Integer;
48    begin
49      delay 1.5;

```

(continues on next page)

(continued from previous page)

```
50      while not Q_I.Empty loop
51          Q_I.Pop (V);
52          Put_Line ("Value from queue: " & Integer'Image (V));
53          delay 0.2;
54      end loop;
55  end T_Consumer;
56 begin
57     null;
58 end Concurrent_Test;
59
60 end Queue_Tests;
```

Listing 11: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Queue_Tests;      use Queue_Tests;
5
6  procedure Main is
7      type Test_Case_Index is (Simple_Queue_Chk,
8                                Concurrent_Queue_Chk);
9
10 procedure Check (TC : Test_Case_Index) is
11
12 begin
13     case TC is
14         when Simple_Queue_Chk =>
15             Simple_Test;
16         when Concurrent_Queue_Chk =>
17             Concurrent_Test;
18     end case;
19 end Check;
20
21 begin
22     if Argument_Count < 1 then
23         Put_Line ("ERROR: missing arguments! Exiting..."); 
24         return;
25     elsif Argument_Count > 1 then
26         Put_Line ("Ignoring additional arguments..."); 
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)));
30 end Main;
```

DESIGN BY CONTRACTS

98.1 Price Range

Goal: use predicates to indicate the correct range of prices.

Steps:

1. Complete the Prices package.
 1. Rewrite the type declaration of Price.

Requirements:

1. Type Price must use a predicate instead of a range.

Remarks:

1. As discussed in the course, ranges are a form of contract.
 1. For example, the subtype Price below indicates that a value of this subtype must always be positive:

```
subtype Price is Amount range 0.0 .. Amount'Last;
```

2. Interestingly, you can replace ranges by predicates, which is the goal of this exercise.

Listing 1: prices.ads

```
1 package Prices is
2
3     type Amount is delta 10.0 ** (-2) digits 12;
4
5     subtype Price is Amount range 0.0 .. Amount'Last;
6
7 end Prices;
```

Listing 2: main.adb

```
1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Prices;           use Prices;
6
7 procedure Main is
8
9     type Test_Case_Index is
10        (Price_Range_Chk);
11
```

(continues on next page)

(continued from previous page)

```

12  procedure Check (TC : Test_Case_Index) is
13
14      procedure Check_Range (A : Amount) is
15          P : constant Price := A;
16      begin
17          Put_Line ("Price: " & Price'Image (P));
18      end Check_Range;
19
20  begin
21      case TC is
22          when Price_Range_Chk =>
23              Check_Range (-2.0);
24      end case;
25  exception
26      when Constraint_Error =>
27          Put_Line ("Constraint_Error detected (NOT as expected).");
28      when Assert_Failure =>
29          Put_Line ("Assert_Failure detected (as expected).");
30  end Check;
31
32 begin
33     if Argument_Count < 1 then
34         Put_Line ("ERROR: missing arguments! Exiting... ");
35         return;
36     elsif Argument_Count > 1 then
37         Put_Line ("Ignoring additional arguments... ");
38     end if;
39
40     Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

98.2 Pythagorean Theorem: Predicate

Goal: use the Pythagorean theorem as a predicate.

Steps:

1. Complete the Triangles package.
 1. Add a predicate to the Right_Triangle type.

Requirements:

1. The Right_Triangle type must use the Pythagorean theorem as a predicate to ensure that its components are consistent.

Remarks:

1. As you probably remember, the Pythagoras' theorem¹³⁶ states that the square of the hypotenuse of a right triangle is equal to the sum of the squares of the other two sides.

Listing 3: triangles.ads

```

1  package Triangles is
2
3      subtype Length is Integer;
4
5      type Right_Triangle is record

```

(continues on next page)

¹³⁶ https://en.wikipedia.org/wiki/Pythagorean_theorem

(continued from previous page)

```

6      H      : Length := 0;
7      -- Hypotenuse
8      C1, C2 : Length := 0;
9      -- Catheti / legs
10     end record;
11
12    function Init (H, C1, C2 : Length) return Right_Triangle is
13      ((H, C1, C2));
14
15  end Triangles;

```

Listing 4: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 5: triangles-io.adb

```

1 package body Triangles.IO is
2
3   function Image (T : Right_Triangle) return String is
4     ("(" & Length'Image (T.H)
5      & ", " & Length'Image (T.C1)
6      & ", " & Length'Image (T.C2)
7      & ")");
8
9 end Triangles.IO;

```

Listing 6: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10  type Test_Case_Index is
11    (Triangle_8_6_Pass_Chk,
12     Triangle_8_6_Fail_Chk,
13     Triangle_10_24_Pass_Chk,
14     Triangle_10_24_Fail_Chk,
15     Triangle_18_24_Pass_Chk,
16     Triangle_18_24_Fail_Chk);
17
18  procedure Check (TC : Test_Case_Index) is
19
20    procedure Check_Triangle (H, C1, C2 : Length) is
21      T : Right_Triangle;
22    begin
23      T := Init (H, C1, C2);
24      Put_Line (Image (T));
25    exception
26      when Constraint_Error =>
27        Put_Line ("Constraint_Error detected (NOT as expected).");

```

(continues on next page)

(continued from previous page)

```

28      when Assert_Failure =>
29        Put_Line ("Assert_Failure detected (as expected).");
30  end Check_Triangle;

31
32 begin
33   case TC is
34     when Triangle_8_6_Pass_Chk  => Check_Triangle (10, 8, 6);
35     when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36     when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37     when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38     when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39     when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40   end case;
41 end Check;

42
43 begin
44   if Argument_Count < 1 then
45     Put_Line ("ERROR: missing arguments! Exiting... ");
46     return;
47   elsif Argument_Count > 1 then
48     Put_Line ("Ignoring additional arguments... ");
49   end if;
50
51   Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

98.3 Pythagorean Theorem: Precondition

Goal: use the Pythagorean theorem as a precondition.

Steps:

1. Complete the Triangles package.
 1. Add a precondition to the Init function.

Requirements:

1. The Init function must use the Pythagorean theorem as a precondition to ensure that the input values are consistent.

Remarks:

1. In this exercise, you'll work again with the Right_Triangle type.
 1. This time, your job is to use a precondition instead of a predicate.
 2. The precondition is applied to the Init function, not to the Right_Triangle type.

Listing 7: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is record
6     H      : Length := 0;
7     -- Hypotenuse
8     C1, C2 : Length := 0;
9     -- Catheti / legs
10    end record;

```

(continues on next page)

(continued from previous page)

```

11
12   function Init (H, C1, C2 : Length) return Right_Triangle is
13     ((H, C1, C2));
14
15 end Triangles;

```

Listing 8: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 9: triangles-io.adb

```

1 package body Triangles.IO is
2
3   function Image (T : Right_Triangle) return String is
4     ("(" & Length'Image (T.H)
5      & ", " & Length'Image (T.C1)
6      & ", " & Length'Image (T.C2)
7      & ")");
8
9 end Triangles.IO;

```

Listing 10: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10 type Test_Case_Index is
11   (Triangle_8_6_Pass_Chk,
12    Triangle_8_6_Fail_Chk,
13    Triangle_10_24_Pass_Chk,
14    Triangle_10_24_Fail_Chk,
15    Triangle_18_24_Pass_Chk,
16    Triangle_18_24_Fail_Chk);
17
18 procedure Check (TC : Test_Case_Index) is
19
20   procedure Check_Triangle (H, C1, C2 : Length) is
21     T : Right_Triangle;
22   begin
23     T := Init (H, C1, C2);
24     Put_Line (Image (T));
25   exception
26     when Constraint_Error =>
27       Put_Line ("Constraint_Error detected (NOT as expected).");
28     when Assert_Failure =>
29       Put_Line ("Assert_Failure detected (as expected).");
30   end Check_Triangle;
31
32 begin

```

(continues on next page)

(continued from previous page)

```

33  case TC is
34    when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35    when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36    when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37    when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38    when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39    when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40  end case;
41 end Check;

42
43 begin
44  if Argument_Count < 1 then
45    Put_Line ("ERROR: missing arguments! Exiting...");  

46    return;
47  elsif Argument_Count > 1 then
48    Put_Line ("Ignoring additional arguments...");  

49  end if;
50
51  Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

98.4 Pythagorean Theorem: Postcondition

Goal: use the Pythagorean theorem as a postcondition.

Steps:

1. Complete the Triangles package.
 1. Add a postcondition to the Init function.

Requirements:

1. The Init function must use the Pythagorean theorem as a postcondition to ensure that the returned object is consistent.

Remarks:

1. In this exercise, you'll work again with the Triangles package.
 1. This time, your job is to apply a postcondition instead of a precondition to the Init function.

Listing 11: triangles.ads

```

1  package Triangles is
2
3    subtype Length is Integer;
4
5    type Right_Triangle is record
6      H      : Length := 0;
7      -- Hypotenuse
8      C1, C2 : Length := 0;
9      -- Catheti / legs
10     end record;
11
12    function Init (H, C1, C2 : Length) return Right_Triangle is
13      ((H, C1, C2));
14
15 end Triangles;

```

Listing 12: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 13: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         ("(" & Length'Image (T.H)
5          & ", " & Length'Image (T.C1)
6          & ", " & Length'Image (T.C2)
7          & ")");
8
9 end Triangles.IO;

```

Listing 14: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10    type Test_Case_Index is
11        (Triangle_8_6_Pass_Chk,
12         Triangle_8_6_Fail_Chk,
13         Triangle_10_24_Pass_Chk,
14         Triangle_10_24_Fail_Chk,
15         Triangle_18_24_Pass_Chk,
16         Triangle_18_24_Fail_Chk);
17
18    procedure Check (TC : Test_Case_Index) is
19
20        procedure Check_Triangle (H, C1, C2 : Length) is
21            T : Right_Triangle;
22        begin
23            T := Init (H, C1, C2);
24            Put_Line (Image (T));
25        exception
26            when Constraint_Error =>
27                Put_Line ("Constraint_Error detected (NOT as expected).");
28            when Assert_Failure =>
29                Put_Line ("Assert_Failure detected (as expected).");
30        end Check_Triangle;
31
32    begin
33        case TC is
34            when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35            when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36            when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37            when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38            when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39            when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);

```

(continues on next page)

(continued from previous page)

```

40      end case;
41  end Check;

42
43 begin
44  if Argument_Count < 1 then
45    Put_Line ("ERROR: missing arguments! Exiting..."); 
46    return;
47  elsif Argument_Count > 1 then
48    Put_Line ("Ignoring additional arguments..."); 
49  end if;
50
51  Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

98.5 Pythagorean Theorem: Type Invariant

Goal: use the Pythagorean theorem as a type invariant.

Steps:

1. Complete the Triangles package.
 1. Add a type invariant to the Right_Triangle type.

Requirements:

1. Right_Triangle is a private type.
 1. It must use the Pythagorean theorem as a type invariant to ensure that its encapsulated components are consistent.

Remarks:

1. In this exercise, Right_Triangle is declared as a private type.
 1. In this case, we use a type invariant for Right_Triangle to check the Pythagorean theorem.
2. As a bonus, after completing the exercise, you may analyze the effect that default values have on type invariants.
 1. For example, the declaration of Right_Triangle uses zero as the default values of the three triangle lengths.
 2. If you replace those default values with Length'Last, you'll get different results.
 3. Make sure you understand why this is happening.

Listing 15: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is private;
6
7     function Init (H, C1, C2 : Length) return Right_Triangle;
8
9 private
10
11    type Right_Triangle is record
12        H      : Length := 0;

```

(continues on next page)

(continued from previous page)

```

13      -- Hypotenuse
14      C1, C2 : Length := 0;
15      -- Catheti / legs
16  end record;

17
18  function Init (H, C1, C2 : Length) return Right_Triangle is
19      ((H, C1, C2));
20
21 end Triangles;

```

Listing 16: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 17: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         ("(" & Length'Image (T.H)
5          & ", " & Length'Image (T.C1)
6          & ", " & Length'Image (T.C2)
7          & ")");
8
9 end Triangles.IO;

```

Listing 18: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_Io;       use Ada.Text_Io;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10    type Test_Case_Index is
11        (Triangle_8_6_Pass_Chk,
12         Triangle_8_6_Fail_Chk,
13         Triangle_10_24_Pass_Chk,
14         Triangle_10_24_Fail_Chk,
15         Triangle_18_24_Pass_Chk,
16         Triangle_18_24_Fail_Chk);
17
18    procedure Check (TC : Test_Case_Index) is
19
20        procedure Check_Triangle (H, C1, C2 : Length) is
21            T : Right_Triangle;
22        begin
23            T := Init (H, C1, C2);
24            Put_Line (Image (T));
25        exception
26            when Constraint_Error =>
27                Put_Line ("Constraint_Error detected (NOT as expected).");
28            when Assert_Failure =>

```

(continues on next page)

(continued from previous page)

```

29      Put_Line ("Assert_Failure detected (as expected).");
30  end Check_Triangle;
31
32 begin
33  case TC is
34    when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35    when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36    when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37    when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38    when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39    when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40  end case;
41 end Check;
42
43 begin
44  if Argument_Count < 1 then
45    Put_Line ("ERROR: missing arguments! Exiting...");
46    return;
47  elsif Argument_Count > 1 then
48    Put_Line ("Ignoring additional arguments...");
49  end if;
50
51  Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

98.6 Primary Color

Goal: extend a package for HTML colors so that it can handle primary colors.

Steps:

1. Complete the Color_Types package.
 1. Declare the HTML_RGB_Color subtype.
 2. Implement the To_Int_Color function.

Requirements:

1. The HTML_Color type is an enumeration that contains a list of HTML colors.
2. The To_RGB_Lookup_Table array implements a lookup-table to convert the colors into a hexadecimal value using RGB color components (i.e. Red, Green and Blue)
3. Function To_Int_Color extracts one of the RGB components of an HTML color and returns its hexadecimal value.
 1. The function has two parameters:
 - First parameter is the HTML color (HTML_Color type).
 - Second parameter indicates which RGB component is to be extracted from the HTML color (HTML_RGB_Color subtype).
 2. For example, if we call To_Int_Color (Salmon, Red), the function returns #FA,
 - This is the hexadecimal value of the red component of the Salmon color.
 - You can find further remarks below about this color as an example.
4. The HTML_RGB_Color subtype is limited to the primary RGB colors components (i.e. Red, Green and Blue).
 1. This subtype is used to select the RGB component in calls to To_Int_Color.

2. You must use a predicate in the type declaration.

Remarks:

1. In this exercise, we reuse the code of the Colors: Lookup-Table exercise from the [Arrays](#) (page 1007) labs.
2. These are the hexadecimal values of the colors that we used in the original exercise:

Color	Value
Salmon	#FA8072
Firebrick	#B22222
Red	#FF0000
Darkred	#8B0000
Lime	#00FF00
Forestgreen	#228B22
Green	#008000
Darkgreen	#006400
Blue	#0000FF
Mediumblue	#0000CD
Darkblue	#00008B

3. You can extract the hexadecimal value of each primary color by splitting the values from the table above into three hexadecimal values with two digits each.

- For example, the hexadecimal value of Salmon is #FA8072, where:
 - the first part of this hexadecimal value (#FA) corresponds to the red component,
 - the second part (#80) corresponds to the green component, and
 - the last part (#72) corresponds to the blue component.

Listing 19: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   subtype Int_Color is Integer range 0 .. 255;
17
18   function Image (I : Int_Color) return String;
19
20   type RGB is record
21     Red   : Int_Color;
22     Green : Int_Color;
23     Blue  : Int_Color;
24   end record;
25
26   function To_RGB (C : HTML_Color) return RGB;

```

(continues on next page)

(continued from previous page)

```

27
28   function Image (C : RGB) return String;
29
30   type HTML_Color_RGB_Array is array (HTML_Color) of RGB;
31
32   To_RGB_Lookup_Table : constant HTML_Color_RGB_Array
33     := (Salmon      => (16#FA#, 16#80#, 16#72#),
34          Firebrick    => (16#B2#, 16#22#, 16#22#),
35          Red          => (16#FF#, 16#00#, 16#00#),
36          Darkred      => (16#8B#, 16#00#, 16#00#),
37          Lime          => (16#00#, 16#FF#, 16#00#),
38          Forestgreen  => (16#22#, 16#8B#, 16#22#),
39          Green         => (16#00#, 16#80#, 16#00#),
40          Darkgreen    => (16#00#, 16#64#, 16#00#),
41          Blue          => (16#00#, 16#00#, 16#FF#),
42          Mediumblue   => (16#00#, 16#00#, 16#CD#),
43          Darkblue     => (16#00#, 16#00#, 16#8B#));
44
45   subtype HTML_RGB_Color is HTML_Color;
46
47   function To_Int_Color (C : HTML_Color;
48                         S : HTML_RGB_Color) return Int_Color;
49   -- Convert to hexadecimal value for the selected RGB component S
50
51 end Color_Types;

```

Listing 20: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5    function To_RGB (C : HTML_Color) return RGB is
6    begin
7      return To_RGB_Lookup_Table (C);
8    end To_RGB;
9
10   function To_Int_Color (C : HTML_Color;
11                         S : HTML_RGB_Color) return Int_Color is
12   begin
13     -- Implement function!
14     return 0;
15   end To_Int_Color;
16
17   function Image (I : Int_Color) return String is
18     subtype Str_Range is Integer range 1 .. 10;
19     S : String (Str_Range);
20   begin
21     Ada.Integer_Text_IO.Put (To      => S,
22                             Item    => I,
23                             Base    => 16);
24     return S;
25   end Image;
26
27   function Image (C : RGB) return String is
28   begin
29     return ("(Red => "      & Image (C.Red)
30            & ", Green => " & Image (C.Green)
31            & ", Blue => "  & Image (C.Blue)
32            & ")");
33   end Image;

```

(continues on next page)

(continued from previous page)

```
34
35 end Color_Types;
```

Listing 21: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7    type Test_Case_Index is
8      (HTML_Color_Red_Chk,
9       HTML_Color_Green_Chk,
10      HTML_Color_Blue_Chk);
11
12  procedure Check (TC : Test_Case_Index) is
13
14    procedure Check_HTML_Colors (S : HTML_RGB_Color) is
15    begin
16      Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
17      for I in HTML_Color'Range loop
18        Put_Line (HTML_Color'Image (I) & " => "
19                  & Image (To_Int_Color (I, S)) & ".");
20      end loop;
21    end Check_HTML_Colors;
22
23  begin
24    case TC is
25      when HTML_Color_Red_Chk =>
26        Check_HTML_Colors (Red);
27      when HTML_Color_Green_Chk =>
28        Check_HTML_Colors (Green);
29      when HTML_Color_Blue_Chk =>
30        Check_HTML_Colors (Blue);
31    end case;
32  end Check;
33
34  begin
35    if Argument_Count < 1 then
36      Put_Line ("ERROR: missing arguments! Exiting... ");
37      return;
38    elsif Argument_Count > 1 then
39      Put_Line ("Ignoring additional arguments... ");
40    end if;
41
42    Check (Test_Case_Index'Value (Argument (1)));
43  end Main;
```


OBJECT-ORIENTED PROGRAMMING

99.1 Simple type extension

Goal: work with type extensions using record types containing numeric components.

Steps:

1. Implement the Type_Extensions package.
 1. Declare the record type T_Float.
 2. Declare the record type T_Mixed
 3. Implement the Init function for the T_Float type with a floating-point input parameter.
 4. Implement the Init function for the T_Float type with an integer input parameter.
 5. Implement the Image function for the T_Float type.
 6. Implement the Init function for the T_Mixed type with a floating-point input parameter.
 7. Implement the Init function for the T_Mixed type with an integer input parameter.
 8. Implement the Image function for the T_Mixed type.

Requirements:

1. Record type T_Float contains the following component:
 1. F, a floating-point type.
2. Record type T_Mixed is derived from the T_Float type.
 1. T_Mixed extends T_Float with the following component:
 1. I, an integer component.
 2. Both components must be numerically *synchronized*:
 - For example, if the floating-point component contains the value 2.0, the value of the integer component must be 2.
 - In order to simplify the implementation, you can simply use **Integer** (F) to convert a floating-point variable F to integer.
3. Function Init returns an object of the corresponding type (T_Float or T_Mixed).
 1. For each type, two versions of Init must be declared:
 1. one with a floating-point input parameter,
 2. another with an integer input parameter.
 2. The parameter to Init is used to initialize the record components.

4. Function Image returns a string for the components of the record type.
 1. In case of the Image function for the T_Float type, the string must have the format "`{ F => <float value> }`".
 - For example, the call `Image (T_Float'(Init (8.0)))` should return the string "`{ F => 8.00000E+00 }`".
 2. In case of the Image function for the T_Mixed type, the string must have the format "`{ F => <float value>, I => <integer value> }`".
 - For example, the call `Image (T_Mixed'(Init (8.0)))` should return the string "`{ F => 8.00000E+00, I => 8 }`".

Listing 1: type_extensions.ads

```
1 package Type_Extensions is
2
3     -- Create declaration of T_Float type!
4     type T_Float is null record;
5
6     -- function Init ...
7
8     -- function Image ...
9
10    -- Create declaration of T_Mixed type!
11    type T_Mixed is null record;
12
13 end Type_Extensions;
```

Listing 2: type_extensions.adb

```
1 package body Type_Extensions is
2
3 end Type_Extensions;
```

Listing 3: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Type_Extensions; use Type_Extensions;
5
6 procedure Main is
7
8     type Test_Case_Index is
9         (Type_Extension_Chk);
10
11    procedure Check (TC : Test_Case_Index) is
12        F1, F2 : T_Float;
13        M1, M2 : T_Mixed;
14    begin
15        case TC is
16        when Type_Extension_Chk =>
17            F1 := Init (2.0);
18            F2 := Init (3);
19            M1 := Init (4.0);
20            M2 := Init (5);
21
22        if M2 in T_Float'Class then
23            Put_Line ("T_Mixed is in T_Float'Class as expected");
24        end if;
25    end;
```

(continues on next page)

(continued from previous page)

```

26      Put_Line ("F1: " & Image (F1));
27      Put_Line ("F2: " & Image (F2));
28      Put_Line ("M1: " & Image (M1));
29      Put_Line ("M2: " & Image (M2));
30  end case;
31 end Check;

32
33 begin
34  if Argument_Count < 1 then
35      Put_Line ("ERROR: missing arguments! Exiting...");  

36      return;
37  elsif Argument_Count > 1 then
38      Put_Line ("Ignoring additional arguments...");  

39  end if;
40
41  Check (Test_Case_Index'Value (Argument (1)));
42 end Main;

```

99.2 Online Store

Goal: create an online store for the members of an association.

Steps:

1. Implement the `Online_Store` package.
 1. Declare the `Member` type.
 2. Declare the `Full_Member` type.
 3. Implement the `Get_Status` function for the `Member` type.
 4. Implement the `Get_Price` function for the `Member` type.
 5. Implement the `Get_Status` function for the `Full_Member` type.
 6. Implement the `Get_Price` function for the `Full_Member` type.
2. Implement the `Online_Store.Tests` child package.
 1. Implement the `Simple_Test` procedure.

Requirements:

1. Package `Online_Store` implements an online store application for the members of an association.
 1. In this association, members can have one of the following status:
 - associate member, or
 - full member.
2. Function `Get_Price` returns the correct price of an item.
 1. Associate members must pay the full price when they buy items from the online store.
 2. Full members can get a discount.
 1. The discount rate can be different for each full member — depending on factors that are irrelevant for this exercise.
3. Package `Online_Store` has following types:
 1. Percentage type, which represents a percentage ranging from 0.0 to 1.0.

2. Member type for associate members containing following components:
 - Start, which indicates the starting year of the membership.
 - This information is common for both associate and full members.
 - You can use the Year_Number type from the standard Ada.Calendar package for this component.
3. Full_Member type for full members.
 1. This type must extend the Member type above.
 2. It contains the following additional component:
 - Discount, which indicates the discount rate that the full member gets in the online store.
 - This component must be of Percentage type.
4. For the Member and Full_Member types, you must implement the following functions:
 1. Get_Status, which returns a string with the membership status.
 - The string must be "Associate Member" or "Full Member", respectively.
 2. Get_Price, which returns the *adapted price* of an item — indicating the actual due amount.
 - For example, for a full member with a 10% discount rate, the actual due amount of an item with a price of 100.00 is 90.00.
 - Associated members don't get a discount, so they always pay the full price.
5. Procedure Simple_Test (from the Online_Store.Tests package) is used for testing.
 1. Based on a list of members that bought on the online store and the corresponding full price of the item, Simple_Test must display information about each member and the actual due amount after discounts.
 2. Information about the members must be displayed in the following format:

```
Member # <number>
Status: <status>
Since:  <year>
Due Amount: <value>
-----
```

3. For this exercise, Simple_Test must use the following list:

#	Membership status	Start (year)	Discount	Full Price
1	Associate	2010	N/A	250.00
2	Full	1998	10.0 %	160.00
3	Full	1987	20.0 %	400.00
4	Associate	2013	N/A	110.00

4. In order to pass the tests, the information displayed by a call to Simple_Test must conform to the format described above.
 - You can find another example in the remarks below.

Remarks:

1. In previous labs, we could have implemented a simplified version of the system described above by simply using an enumeration type to specify the membership status. For example:

```
type Member_Status is (Associate_Member, Full_Member);
```

1. In this case, the Get_Price function would then evaluate the membership status and adapt the item price — assuming a fixed discount rate for all full members. This could be the corresponding function declaration:

```
type Amount is delta 10.0**(-2) digits 10;
function Get_Price (M : Member_Status;
P : Amount) return Amount;
```

2. In this exercise, however, we'll use type extension to represent the membership status in our application.
2. For the procedure Simple_Test, let's consider the following list of members as an example:

#	Membership status	Start (year)	Discount	Full Price
1	Associate	2002	N/A	100.00
2	Full	2005	10.0 %	100.00

- For this list, the test procedure displays the following information (in this exact format):

```
Member # 1
Status: Associate Member
Since: 2002
Due Amount: 100.00
-----
Member # 2
Status: Full Member
Since: 2005
Due Amount: 90.00
-----
```

- Here, although both members had the same full price (as indicated by the last column), member #2 gets a reduced due amount of 90.00 because of the full membership status.

Listing 4: online_store.ads

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 package Online_Store is
4
5   type Amount is delta 10.0**(-2) digits 10;
6
7   subtype Percentage is Amount range 0.0 .. 1.0;
8
9   -- Create declaration of Member type!
10
11  -- You can use Year_Number from Ada.Calendar for the membership
12  -- starting year.
13
14  type Member is null record;
15
16  function Get_Status (M : Member) return String;
17
18  function Get_Price (M : Member;
19    P : Amount) return Amount;
```

(continues on next page)

(continued from previous page)

```

21   -- Create declaration of Full_Member type!
22   --
23   -- Use the Percentage type for storing the membership discount.
24   --
25   type Full_Member is null record;
26
27   function Get_Status (M : Full_Member) return String;
28
29   function Get_Price (M : Full_Member;
30                      P : Amount) return Amount;
31
32 end Online_Store;
```

Listing 5: online_store.adb

```

1 package body Online_Store is
2
3   function Get_Status (M : Member) return String is
4     ("");
5
6   function Get_Status (M : Full_Member) return String is
7     ("");
8
9   function Get_Price (M : Member;
10                      P : Amount) return Amount is (0.0);
11
12  function Get_Price (M : Full_Member;
13                      P : Amount) return Amount is
14    (0.0);
15
16 end Online_Store;
```

Listing 6: online_store-tests.ads

```

1 package Online_Store.Tests is
2
3   procedure Simple_Test;
4
5 end Online_Store.Tests;
```

Listing 7: online_store-tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Online_Store.Tests is
4
5   procedure Simple_Test is
6   begin
7     null;
8   end Simple_Test;
9
10 end Online_Store.Tests;
```

Listing 8: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Online_Store;         use Online_Store;
```

(continues on next page)

(continued from previous page)

```

5  with Online_Store.Tests; use Online_Store.Tests;
6
7  procedure Main is
8
9    type Test_Case_Index is
10      (Type_Chk,
11       Unit_Test_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14
15    function Result_Image (Result : Boolean) return String is
16      (if Result then "OK" else "not OK");
17
18  begin
19    case TC is
20      when Type_Chk =>
21        declare
22          AM : constant Member      := (Start      => 2002);
23          FM : constant Full_Member := (Start      => 1990,
24                                         Discount => 0.2);
25        begin
26          Put_Line ("Testing Status of Associate Member Type => "
27                    & Result_Image (AM.Get_Status = "Associate Member"));
28          Put_Line ("Testing Status of Full Member Type => "
29                    & Result_Image (FM.Get_Status = "Full Member"));
30          Put_Line ("Testing Discount of Associate Member Type => "
31                    & Result_Image (AM.Get_Price (100.0) = 100.0));
32          Put_Line ("Testing Discount of Full Member Type => "
33                    & Result_Image (FM.Get_Price (100.0) = 80.0));
34        end;
35        when Unit_Test_Chk =>
36          Simple_Test;
37    end case;
38  end Check;
39
40 begin
41  if Argument_Count < 1 then
42    Put_Line ("ERROR: missing arguments! Exiting... ");
43    return;
44  elsif Argument_Count > 1 then
45    Put_Line ("Ignoring additional arguments... ");
46  end if;
47
48  Check (Test_Case_Index'Value (Argument (1)));
49 end Main;

```


STANDARD LIBRARY: CONTAINERS

100.1 Simple todo list

Goal: implement a simple to-do list system using vectors.

Steps:

1. Implement the Todo_Lists package.
 1. Declare the Todo_Item type.
 2. Declare the Todo_List type.
 3. Implement the Add procedure.
 4. Implement the Display procedure.
2. Todo_Item type is used to store to-do items.
 1. It should be implemented as an access type to strings.
3. Todo_List type is the container for all to-do items.
 1. It should be implemented as a **vector**.
4. Procedure Add adds items (of Todo_Item type) to the list (of Todo_List type).
 1. This requires allocating a string for the access type.
5. Procedure Display is used to display all to-do items.
 1. It must display one item per line.

Remarks:

1. This exercise is based on the *Simple todo list* exercise from the [More About Types](#) (page 1025).
 1. Your goal is to rewrite that exercise using vectors instead of arrays.
 2. You may reuse the code you've already implemented as a starting point.

Listing 1: todo_lists.ads

```
1 package Todo_Lists is
2
3     type Todo_Item is access String;
4
5     type Todo_List is null record;
6
7     procedure Add (Todos : in out Todo_List;
8                     Item   : String);
9
10    procedure Display (Todos : Todo_List);
11
12 end Todo_Lists;
```

Listing 2: todo_lists.adb

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  package body Todo_Lists is
4
5      procedure Add (Todos : in out Todo_List;
6                      Item   : String) is
7          begin
8              null;
9          end Add;
10
11     procedure Display (Todos : Todo_List) is
12         begin
13             Put_Line ("TO-DO LIST");
14         end Display;
15
16 end Todo_Lists;
```

Listing 3: main.adb

```
1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_Io;        use Ada.Text_Io;
3
4  with Todo_Lists;        use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List;
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreasheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27         end case;
28     end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting... ");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments... ");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;
```

100.2 List of unique integers

Goal: create function that removes duplicates from and orders a collection of elements.

Steps:

1. Implement package Ops.
 1. Declare the Int_Array type.
 2. Declare the Integer_Sets type.
 3. Implement the Get_Uncue function that returns a set.
 4. Implement the Get_Uncue function that returns an array of integer values.

Requirements:

1. The Int_Array type is an unconstrained array of positive range.
2. The Integer_Sets package is an instantiation of the Ordered_Sets package for the **Integer** type.
3. The Get_Uncue function must remove duplicates from an input array of integer values and order the elements.
 1. For example:
 - if the input array contains (7, 7, 1)
 - the function must return (1, 7).
2. You must implement this function by using sets from the Ordered_Sets package.
3. Get_Uncue must be implemented in two versions:
 - one version that returns a set — Set type from the Ordered_Sets package.
 - one version that returns an array of integer values — Int_Array type.

Remarks:

1. Sets — as the one found in the generic Ordered_Sets package — are useful for quickly and easily creating an algorithm that removes duplicates from a list of elements.

Listing 4: ops.ads

```

1  with Ada.Containers.Ordered_Sets;
2
3  package Ops is
4
5    -- type Int_Array is ...
6
7    -- package Integer_Sets is ...
8
9    subtype Int_Set is Integer_Sets.Set;
10
11   function Get_Uncue (A : Int_Array) return Int_Set;
12
13   function Get_Uncue (A : Int_Array) return Int_Array;
14
15 end Ops;

```

Listing 5: ops.adb

```

1  package body Ops is
2
3    function Get_Uncue (A : Int_Array) return Int_Set is

```

(continues on next page)

(continued from previous page)

```

4 begin
5   null;
6 end Get_Undup;
7
8 function Get_Undup (A : Int_Array) return Int_Array is
9 begin
10   null;
11 end Get_Undup;
12
13 end Ops;

```

Listing 6: main.adb

```

1 with Ada.Command_Line;           use Ada.Command_Line;
2 with Ada.Text_IO;               use Ada.Text_IO;
3
4 with Ops;                      use Ops;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Get_Undup_Set_Ck,
9      Get_Undup_Array_Ck);
10
11 procedure Check (TC : Test_Case_Index;
12                   A : Int_Array) is
13
14   procedure Display_Undup_Set (A : Int_Array) is
15     S : constant Int_Set := Get_Undup (A);
16   begin
17     for E of S loop
18       Put_Line (Integer'Image (E));
19     end loop;
20   end Display_Undup_Set;
21
22   procedure Display_Undup_Array (A : Int_Array) is
23     AU : constant Int_Array := Get_Undup (A);
24   begin
25     for E of AU loop
26       Put_Line (Integer'Image (E));
27     end loop;
28   end Display_Undup_Array;
29
30 begin
31   case TC is
32     when Get_Undup_Set_Ck => Display_Undup_Set (A);
33     when Get_Undup_Array_Ck => Display_Undup_Array (A);
34   end case;
35 end Check;
36
37 begin
38   if Argument_Count < 3 then
39     Put_Line ("ERROR: missing arguments! Exiting...");
40     return;
41   else
42     declare
43       A : Int_Array (1 .. Argument_Count - 1);
44     begin
45       for I in A'Range loop
46         A (I) := Integer'Value (Argument (1 + I));
47       end loop;
48       Check (Test_Case_Index'Value (Argument (1)), A);

```

(continues on next page)

(continued from previous page)

```
49      end;  
50  end if;  
51 end Main;
```


STANDARD LIBRARY: DATES & TIMES

101.1 Holocene calendar

Goal: create a function that returns the year in the Holocene calendar.

Steps:

1. Implement the To_Holocene_Year function.

Requirements:

1. The To_Holocene_Year extracts the year from a time object (Time type) and returns the corresponding year for the Holocene calendar¹³⁷.
 1. For positive (AD) years, the Holocene year is calculated by adding 10,000 to the year number.

Remarks:

1. In this exercise, we don't deal with BC years.
2. Note that the year component of the Time type from the Ada.Calendar package is limited to years starting with 1901.

Listing 1: to_holocene_year.adb

```
1 with Ada.Calendar; use Ada.Calendar;
2
3 function To_Holocene_Year (T : Time) return Integer is
4 begin
5     return 0;
6 end To_Holocene_Year;
```

Listing 2: main.adb

```
1 with Ada.Command_Line;           use Ada.Command_Line;
2 with Ada.Text_IO;               use Ada.Text_IO;
3 with Ada.Calendar;              use Ada.Calendar;
4
5 with To_Holocene_Year;
6
7 procedure Main is
8     type Test_Case_Index is
9         (Holocene_Chk);
10
11    procedure Display_Holocene_Year (Y : Year_Number) is
12        HY : Integer;
13    begin
```

(continues on next page)

¹³⁷ https://en.wikipedia.org/wiki/Holocene_calendar

(continued from previous page)

```

14    HY := To_Holocene_Year (Time_0f (Y, 1, 1));
15    Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));
16    Put_Line ("Year (Holocene): " & Integer'Image (HY));
17  end Display_Holocene_Year;

18
19  procedure Check (TC : Test_Case_Index) is
20  begin
21    case TC is
22      when Holocene_Chk =>
23        Display_Holocene_Year (2012);
24        Display_Holocene_Year (2020);
25    end case;
26  end Check;

27
28 begin
29  if Argument_Count < 1 then
30    Put_Line ("ERROR: missing arguments! Exiting... ");
31    return;
32  elsif Argument_Count > 1 then
33    Put_Line ("Ignoring additional arguments... ");
34  end if;
35
36  Check (Test_Case_Index'Value (Argument (1)));
37 end Main;

```

101.2 List of events

Goal: create a system to manage a list of events.

Steps:

1. Implement the Events package.
 1. Declare the Event_Item type.
 2. Declare the Event_Items type.
2. Implement the Events.Lists package.
 1. Declare the Event_List type.
 2. Implement the Add procedure.
 3. Implement the Display procedure.

Requirements:

1. The Event_Item type (from the Events package) contains the *description of an event*.
 1. This description shall be stored in an access-to-string type.
2. The Event_Items type stores a list of events.
 1. This will be used later to represent multiple events for a specific date.
 2. You shall use a vector for this type.
3. The Events.Lists package contains the subprograms that are used in the test application.
4. The Event_List type (from the Events.Lists package) maps a list of events to a specific date.
 1. You must use the Event_Items type for the list of events.

2. You shall use the Time type from the Ada.Calendar package for the dates.
3. Since we expect the events to be ordered by the date, you shall use ordered maps for the Event_List type.
5. Procedure Add adds an event into the list of events for a specific date.
6. Procedure Display must display all events for each date (ordered by date) using the following format:

```
<event_date #1>
  <description of item #1a>
  <description of item #1b>
<event_date #2>
  <description of item #2a>
  <description of item #2b>
```

1. You should use the auxiliary Date_Image function — available in the body of the Events.Lists package — to display the date in the YYYY-MM-DD format.

Remarks:

1. Let's briefly illustrate the expected output of this system.

1. Consider the following example:

```
with Ada.Calendar;
with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;

with Events.Lists;           use Events.Lists;

procedure Test is
  EL : Event_List;
begin
  EL.Add (Time_Of (2019, 4, 16),
          "Item #2");
  EL.Add (Time_Of (2019, 4, 15),
          "Item #1");
  EL.Add (Time_Of (2019, 4, 16),
          "Item #3");
  EL.Display;
end Test;
```

2. The expected output of the Test procedure must be:

```
EVENTS LIST
- 2019-04-15
  - Item #1
- 2019-04-16
  - Item #2
  - Item #3
```

Listing 3: events.ads

```
1 package Events is
2
3   type Event_Item is null record;
4
5   type Event_Items is null record;
6
7 end Events;
```

Listing 4: events-lists.ads

```
1  with Ada.Calendar; use Ada.Calendar;
2
3  package Events.Lists is
4
5    type Event_List is tagged private;
6
7    procedure Add (Events      : in out Event_List;
8                  Event_Time :        Time;
9                  Event      :        String);
10
11   procedure Display (Events : Event_List);
12
13  private
14
15    type Event_List is tagged null record;
16
17 end Events.Lists;
```

Listing 5: events-lists.adb

```
1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4  package body Events.Lists is
5
6    procedure Add (Events      : in out Event_List;
7                  Event_Time :        Time;
8                  Event      :        String) is
9    begin
10      null;
11    end Add;
12
13    function Date_Image (T : Time) return String is
14      Date_Img : constant String := Image (T);
15    begin
16      return Date_Img (1 .. 10);
17    end;
18
19    procedure Display (Events : Event_List) is
20      T : Time;
21    begin
22      Put_Line ("EVENTS LIST");
23      -- You should use Date_Image (T) here!
24    end Display;
25
26 end Events.Lists;
```

Listing 6: main.adb

```
1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5
6  with Events.Lists;          use Events.Lists;
7
8  procedure Main is
9    type Test_Case_Index is
10      (Event_List_Chk);
```

(continues on next page)

(continued from previous page)

```
11
12 procedure Check (TC : Test_Case_Index) is
13     EL : Event_List;
14 begin
15     case TC is
16         when Event_List_Chk =>
17             EL.Add (Time_Of (2018, 2, 16),
18                     "Final check");
19             EL.Add (Time_Of (2018, 2, 16),
20                     "Release");
21             EL.Add (Time_Of (2018, 12, 3),
22                     "Brother's birthday");
23             EL.Add (Time_Of (2018, 1, 1),
24                     "New Year's Day");
25             EL.Display;
26     end case;
27 end Check;
28
29 begin
30     if Argument_Count < 1 then
31         Put_Line ("ERROR: missing arguments! Exiting...");  

32         return;
33     elsif Argument_Count > 1 then
34         Put_Line ("Ignoring additional arguments...");  

35     end if;
36
37     Check (Test_Case_Index'Value (Argument (1)));
38 end Main;
```


STANDARD LIBRARY: STRINGS

102.1 Concatenation

Goal: implement functions to concatenate an array of unbounded strings.

Steps:

1. Implement the Str_Concat package.
 1. Implement the Concat function for Unbounded_String.
 2. Implement the Concat function for **String**.

Requirements:

1. The first Concat function receives an unconstrained array of unbounded strings and returns the concatenation of those strings as an unbounded string.
 1. The second Concat function has the same parameters, but returns a standard string (**String** type).
2. Both Concat functions have the following parameters:
 1. An unconstrained array of Unbounded_String strings (Unbounded_Strings type).
 2. Trim_Str, a Boolean parameter indicating whether each unbounded string must be trimmed.
 3. Add_Witespace, a Boolean parameter indicating whether a whitespace shall be added between each unbounded string and the next one.
 1. No whitespace shall be added after the last string of the array.

Remarks:

1. You can use the Trim function from the Ada.Strings.Unbounded package.

Listing 1: str_concat.ads

```
1 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2
3 package Str_Concat is
4
5   type Unbounded_Strings is array (Positive range <>) of Unbounded_String;
6
7   function Concat (USA           : Unbounded_Strings;
8                   Trim_Str      : Boolean;
9                   Add_Witespace : Boolean) return Unbounded_String;
10
11  function Concat (USA           : Unbounded_Strings;
12                   Trim_Str      : Boolean;
13                   Add_Witespace : Boolean) return String;
```

(continues on next page)

(continued from previous page)

```
14
15 end Str_Concat;
```

Listing 2: str_concat.adb

```
1 with Ada.Strings; use Ada.Strings;
2
3 package body Str_Concat is
4
5   function Concat (USA          : Unbounded_Strings;
6                     Trim_Str     : Boolean;
7                     Add_Whitespace : Boolean) return Unbounded_String is
8   begin
9     return "";
10  end Concat;
11
12  function Concat (USA          : Unbounded_Strings;
13                     Trim_Str     : Boolean;
14                     Add_Whitespace : Boolean) return String is
15  begin
16    return "";
17  end Concat;
18
19 end Str_Concat;
```

Listing 3: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
4
5 with Str_Concat;           use Str_Concat;
6
7 procedure Main is
8   type Test_Case_Index is
9     (Unbounded_Concat_No_Trim_No_WS_Chk,
10      Unbounded_Concat_Trim_No_WS_Chk,
11      String_Concat_Trim_WS_Chk,
12      Concat_Single_Element);
13
14   procedure Check (TC : Test_Case_Index) is
15   begin
16     case TC is
17       when Unbounded_Concat_No_Trim_No_WS_Chk =>
18         declare
19           S : constant Unbounded_Strings := (
20             To_Unbounded_String ("Hello"),
21             To_Unbounded_String (" World"),
22             To_Unbounded_String ("!"));
23         begin
24           Put_Line (To_String (Concat (S, False, False)));
25         end;
26       when Unbounded_Concat_Trim_No_WS_Chk =>
27         declare
28           S : constant Unbounded_Strings := (
29             To_Unbounded_String (" This "),
30             To_Unbounded_String (" _is_ "),
31             To_Unbounded_String (" a "),
32             To_Unbounded_String (" _check "));
33         begin
```

(continues on next page)

(continued from previous page)

```

34      Put_Line (To_String (Concat (S, True, False)));
35  end;
36  when String_Concat_Trim_WS_Chk =>
37    declare
38      S : constant Unbounded_Strings := (
39          To_Unbounded_String (" This "),
40          To_Unbounded_String (" is a "),
41          To_Unbounded_String (" test. "));
42    begin
43        Put_Line (Concat (S, True, True));
44    end;
45  when Concat_Single_Element =>
46    declare
47      S : constant Unbounded_Strings := (
48          1 => To_Unbounded_String (" Hi "));
49    begin
50        Put_Line (Concat (S, True, True));
51    end;
52  end case;
53 end Check;
54
55 begin
56  if Argument_Count < 1 then
57    Put_Line ("ERROR: missing arguments! Exiting...");  

58    return;
59  elsif Argument_Count > 1 then
60    Put_Line ("Ignoring additional arguments...");  

61  end if;
62
63  Check (Test_Case_Index'Value (Argument (1)));
64 end Main;

```

102.2 List of events

Goal: create a system to manage a list of events.

Steps:

1. Implement the Events package.
 1. Declare the Event_Item subtype.
2. Implement the Events.Lists package.
 1. Adapt the Add procedure.
 2. Adapt the Display procedure.

Requirements:

1. The Event_Item type (from the Events package) contains the *description of an event*.
 1. This description is declared as a subtype of unbounded string.
2. Procedure Add adds an event into the list of events for a specific date.
 1. The declaration of E needs to be adapted to use unbounded strings.
3. Procedure Display must display all events for each date (ordered by date) using the following format:
 1. The arguments to Put_Line need to be adapted to use unbounded strings.

Remarks:

1. We use the lab on the list of events from the previous chapter (*Standard library: Dates & Times* (page 1097)) as a starting point.

Listing 4: events.ads

```

1  with Ada.Containers.Vectors;
2
3  package Events is
4
5    -- subtype Event_Item is
6
7    package Event_ItemContainers is new
8      Ada.Containers.Vectors
9        (Index_Type    => Positive,
10         Element_Type => Event_Item);
11
12   subtype Event_Items is Event_ItemContainers.Vector;
13
14 end Events;

```

Listing 5: events-lists.ads

```

1  with Ada.Calendar;           use Ada.Calendar;
2  with Ada.Containers.Ordered_Maps;
3
4  package Events.Lists is
5
6    type Event_List is tagged private;
7
8    procedure Add (Events      : in out Event_List;
9                  Event_Time : Time;
10                 Event     : String);
11
12   procedure Display (Events : Event_List);
13
14 private
15
16   package Event_Time_ItemContainers is new
17     Ada.Containers.Ordered_Maps
18       (Key_Type      => Time,
19        Element_Type  => Event_Items,
20         "="          => Event_ItemContainers. "=");
21
22   type Event_List is new Event_Time_ItemContainers.Map with null record;
23
24 end Events.Lists;

```

Listing 6: events-lists.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4  package body Events.Lists is
5
6    procedure Add (Events      : in out Event_List;
7                  Event_Time : Time;
8                  Event     : String) is
9      use Event_ItemContainers;
10     E : constant Event_Item := new String'(Event);
11
12 begin
13   if not Events.Contains (Event_Time) then
14     Events.Include (Event_Time, Empty_Vector);

```

(continues on next page)

(continued from previous page)

```

14  end if;
15  Events (Event_Time).Append (E);
16 end Add;

17
18 function Date_Image (T : Time) return String is
19   Date_Img : constant String := Image (T);
20 begin
21   return Date_Img (1 .. 10);
22 end;

23
24 procedure Display (Events : Event_List) is
25   use Event_Time_Item_Containers;
26   T : Time;
27 begin
28   Put_Line ("EVENTS LIST");
29   for C in Events.Iterate loop
30     T := Key (C);
31     Put_Line ("- " & Date_Image (T));
32     for I of Events (C) loop
33       Put_Line ("      - " & I.all);
34     end loop;
35   end loop;
36 end Display;
37
38 end Events.Lists;

```

Listing 7: main.adb

```

1  with Ada.Command_Line;           use Ada.Command_Line;
2  with Ada.Text_Io;               use Ada.Text_Io;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting;  use Ada.Calendar.Formatting;
5  with Ada.Strings.Unbounded;    use Ada.Strings.Unbounded;
6
7  with Events;
8  with Events.Lists;             use Events.Lists;
9
10 procedure Main is
11   type Test_Case_Index is
12     (Unbounded_String_Chk,
13      Event_List_Chk);
14
15 procedure Check (TC : Test_Case_Index) is
16   EL : Event_List;
17 begin
18   case TC is
19     when Unbounded_String_Chk =>
20       declare
21         S : constant Events.Event_Item := To_Unbounded_String ("Checked");
22       begin
23         Put_Line (To_String (S));
24       end;
25     when Event_List_Chk =>
26       EL.Add (Time_Of (2018, 2, 16),
27               "Final check");
28       EL.Add (Time_Of (2018, 2, 16),
29               "Release");
30       EL.Add (Time_Of (2018, 12, 3),
31               "Brother's birthday");
32       EL.Add (Time_Of (2018, 1, 1),
33               "New Year's Day");

```

(continues on next page)

(continued from previous page)

```
34      EL.Display;
35  end case;
36 end Check;

37
38 begin
39  if Argument_Count < 1 then
40    Put_Line ("ERROR: missing arguments! Exiting... ");
41    return;
42  elsif Argument_Count > 1 then
43    Put_Line ("Ignoring additional arguments... ");
44  end if;
45
46  Check (Test_Case_Index'Value (Argument (1)));
47 end Main;
```

STANDARD LIBRARY: NUMERICS

103.1 Decibel Factor

Goal: implement functions to convert from Decibel values to factors and vice-versa.

Steps:

1. Implement the Decibels package.
 1. Implement the To_Decibel function.
 2. Implement the To_Factor function.

Requirements:

1. The subtypes Decibel and Factor are based on a floating-point type.
2. Function To_Decibel converts a multiplication factor (or ratio) to decibels.
 - For the implementation, use $20 * \log_{10}(F)$, where F is the factor/ratio.
3. Function To_Factor converts a value in decibels to a multiplication factor (or ratio).
 - For the implementation, use $10^{D/20}$, where D is the value in Decibel.

Remarks:

1. The Decibel¹³⁸ is used to express the ratio of two values on a logarithmic scale.
 1. For example, an increase of 6 dB corresponds roughly to a multiplication by two (or an increase by 100 % of the original value).
2. You can find the functions that you'll need for the calculation in the Ada.Numerics.Elementary_Functions package.

Listing 1: decibels.ads

```
1 package Decibels is
2
3     subtype Decibel is Float;
4     subtype Factor   is Float;
5
6     function To_Decibel (F : Factor) return Decibel;
7
8     function To_Factor  (D : Decibel) return Factor;
9
10 end Decibels;
```

¹³⁸ <https://en.wikipedia.org/wiki/Decibel>

Listing 2: decibels.adb

```

1 package body Decibels is
2
3     function To_Decibel (F : Factor) return Decibel is
4 begin
5     return 0.0;
6 end To_Decibel;
7
8     function To_Factor (D : Decibel) return Factor is
9 begin
10    return 0.0;
11 end To_Factor;
12
13 end Decibels;

```

Listing 3: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Decibels;         use Decibels;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Db_Chk,
9          Factor_Chk);
10
11 procedure Check (TC : Test_Case_Index; V : Float) is
12
13     package F_IO is new Ada.Text_IO.Float_IO (Factor);
14     package D_IO is new Ada.Text_IO.Float_IO (Decibel);
15
16     procedure Put_Decibel_Cnvt (D : Decibel) is
17         F : constant Factor := To_Factor (D);
18     begin
19         D_IO.Put (D, 0, 2, 0);
20         Put (" dB => Factor of ");
21         F_IO.Put (F, 0, 2, 0);
22         New_Line;
23     end;
24
25     procedure Put_Factor_Cnvt (F : Factor) is
26         D : constant Decibel := To_Decibel (F);
27     begin
28         Put ("Factor of ");
29         F_IO.Put (F, 0, 2, 0);
30         Put (" => ");
31         D_IO.Put (D, 0, 2, 0);
32         Put_Line (" dB");
33     end;
34
35 begin
36     case TC is
37         when Db_Chk =>
38             Put_Decibel_Cnvt (Decibel (V));
39         when Factor_Chk =>
40             Put_Factor_Cnvt (Factor (V));
41     end case;
42 end Check;
43
44 begin

```

(continues on next page)

(continued from previous page)

```

44  if Argument_Count < 2 then
45      Put_Line ("ERROR: missing arguments! Exiting..."); 
46      return;
47  elsif Argument_Count > 2 then
48      Put_Line ("Ignoring additional arguments..."); 
49  end if;
50
51  Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
52 end Main;

```

103.2 Root-Mean-Square

Goal: implement a function to calculate the root-mean-square of a sequence of values.

Steps:

1. Implement the Signals package.
 1. Implement the Rms function.

Requirements:

1. Subtype Sig_Value is based on a floating-point type.
2. Type Signal is an unconstrained array of Sig_Value elements.
3. Function Rms calculates the RMS of a sequence of values stored in an array of type Signal.
 1. See the remarks below for a description of the RMS calculation.

Remarks:

1. The root-mean-square¹³⁹ (RMS) value is an important information associated with sequences of values.
 1. It's used, for example, as a measurement for signal processing.
 2. It is calculated by:
 1. Creating a sequence S with the square of each value of an input sequence S_{in} .
 2. Calculating the mean value M of the sequence S .
 3. Calculating the square-root R of M .
 3. You can optimize the algorithm above by combining steps #1 and #2 into a single step.

Listing 4: signals.ads

```

1 package Signals is
2
3     subtype Sig_Value is Float;
4
5     type Signal is array (Natural range <>) of Sig_Value;
6
7     function Rms (S : Signal) return Sig_Value;
8
9 end Signals;

```

¹³⁹ https://en.wikipedia.org/wiki/Root_mean_square

Listing 5: signals.adb

```

1  with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3  package body Signals is
4
5      function Rms (S : Signal) return Sig_Value is
6      begin
7          return 0.0;
8      end;
9
10 end Signals;

```

Listing 6: signals-std.ads

```

1  package Signals.Std is
2
3      Sample_Rate : Float := 8000.0;
4
5      function Generate_Sine (N : Positive; Freq : Float) return Signal;
6
7      function Generate_Square (N : Positive) return Signal;
8
9      function Generate_Triangular (N : Positive) return Signal;
10
11 end Signals.Std;

```

Listing 7: signals-std.adb

```

1  with Ada.Numerics;                                     use Ada.Numerics;
2  with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
3
4  package body Signals.Std is
5
6      function Generate_Sine (N : Positive; Freq : Float) return Signal is
7          S : Signal (0 .. N - 1);
8      begin
9          for I in S'First .. S'Last loop
10              S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
11          end loop;
12
13      return S;
14  end;
15
16      function Generate_Square (N : Positive) return Signal is
17          S : constant Signal (0 .. N - 1) := (others => 1.0);
18      begin
19          return S;
20      end;
21
22      function Generate_Triangular (N : Positive) return Signal is
23          S      : Signal (0 .. N - 1);
24          S_Half : constant Natural := S'Last / 2;
25      begin
26          for I in S'First .. S_Half loop
27              S (I) := 1.0 * (Float (I) / Float (S_Half));
28          end loop;
29          for I in S_Half .. S'Last loop
30              S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
31          end loop;
32

```

(continues on next page)

(continued from previous page)

```

33      return S;
34  end;
35
36 end Signals.Std;
```

Listing 8: main.adb

```

1  with Ada.Command_Line;           use Ada.Command_Line;
2  with Ada.Text_IO;               use Ada.Text_IO;
3
4  with Signals;                  use Signals;
5  with Signals.Std;              use Signals.Std;
6
7  procedure Main is
8    type Test_Case_Index is
9      (Sine_Signal_Chk,
10       Square_Signal_Chk,
11       Triangular_Signal_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14    package Sig_I0 is new Ada.Text_IO.Float_IO (Sig_Value);
15
16    N      : constant Positive := 1024;
17    S_Si : constant Signal := Generate_Sine (N, 440.0);
18    S_Sq : constant Signal := Generate_Square (N);
19    S_Tr : constant Signal := Generate_Triangular (N + 1);
20
21 begin
22   case TC is
23     when Sine_Signal_Chk =>
24       Put ("RMS of Sine Signal: ");
25       Sig_I0.Put (Rms (S_Si), 0, 2, 0);
26       New_Line;
27     when Square_Signal_Chk =>
28       Put ("RMS of Square Signal: ");
29       Sig_I0.Put (Rms (S_Sq), 0, 2, 0);
30       New_Line;
31     when Triangular_Signal_Chk =>
32       Put ("RMS of Triangular Signal: ");
33       Sig_I0.Put (Rms (S_Tr), 0, 2, 0);
34       New_Line;
35   end case;
36 end Check;
37
38 begin
39   if Argument_Count < 1 then
40     Put_Line ("ERROR: missing arguments! Exiting... ");
41     return;
42   elsif Argument_Count > 1 then
43     Put_Line ("Ignoring additional arguments... ");
44   end if;
45
46   Check (Test_Case_Index'Value (Argument (1)));
47 end Main;
```

103.3 Rotation

Goal: use complex numbers to calculate the positions of an object in a circle after rotation.

Steps:

1. Implement the Rotation package.
 1. Implement the Rotation function.

Requirements:

1. Type Complex_Points is an unconstrained array of complex values.
2. Function Rotation returns a list of positions (represented by the Complex_Points type) when dividing a circle in N equal slices.
 1. See the remarks below for a more detailed explanation.
2. You must use functions from Ada.Numerics.Complex_Types to implement Rotation.
3. Subtype Angle is based on a floating-point type.
4. Type Angles is an unconstrained array of angles.
5. Function To_Angles returns a list of angles based on an input list of positions.

Remarks:

1. Complex numbers are particularly useful in computer graphics to simplify the calculation of rotations.
 1. For example, let's assume you've drawn an object on your screen on position (1.0, 0.0).
 2. Now, you want to move this object in a circular path — i.e. make it rotate around position (0.0, 0.0) on your screen.
 - You could use *sine* and *cosine* functions to calculate each position of the path.
 - However, you could also calculate the positions using complex numbers.
2. In this exercise, you'll use complex numbers to calculate the positions of an object that starts on zero degrees — on position (1.0, 0.0) — and rotates around (0.0, 0.0) for N slices of a circle.
 1. For example, if we divide the circle in four slices, the object's path will consist of following points / positions:

```
Point #1: ( 1.0,  0.0)
Point #2: ( 0.0,  1.0)
Point #3: (-1.0,  0.0)
Point #4: ( 0.0, -1.0)
Point #5: ( 1.0,  0.0)
```

1. As expected, point #5 is equal to the starting point (point #1), since the object rotates around (0.0, 0.0) and returns to the starting point.
2. We can also describe this path in terms of angles. The following list presents the angles for the path on a four-sliced circle:

```
Point #1:    0.00 degrees
Point #2:   90.00 degrees
Point #3: 180.00 degrees
Point #4: -90.00 degrees (= 270 degrees)
Point #5:    0.00 degrees
```

1. To rotate a complex number simply multiply it by a unit vector whose arg is the radian angle to be rotated: $Z = e^{\frac{2\pi}{N}}$

Listing 9: rotation.ads

```

1  with Ada.Numerics.Complex_Types;
2  use Ada.Numerics.Complex_Types;
3
4  package Rotation is
5
6    type Complex_Points is array (Positive range <>) of Complex;
7
8    function Rotation (N : Positive) return Complex_Points;
9
10 end Rotation;

```

Listing 10: rotation.adb

```

1  with Ada.Numerics; use Ada.Numerics;
2
3  package body Rotation is
4
5    function Rotation (N : Positive) return Complex_Points is
6      C : Complex_Points (1 .. 1) := (others => (0.0, 0.0));
7    begin
8      return C;
9    end;
10
11 end Rotation;

```

Listing 11: angles.ads

```

1  with Rotation; use Rotation;
2
3  package Angles is
4
5    subtype Angle is Float;
6
7    type Angles is array (Positive range <>) of Angle;
8
9    function To_Angles (C : Complex_Points) return Angles;
10
11 end Angles;

```

Listing 12: angles.adb

```

1  with Ada.Numerics;           use Ada.Numerics;
2  with Ada.Numerics.Complex_Types; use Ada.Numerics.Complex_Types;
3
4  package body Angles is
5
6    function To_Angles (C : Complex_Points) return Angles is
7    begin
8      return A : Angles (C'Range) do
9        for I in A'Range loop
10          A (I) := Argument (C (I)) / Pi * 180.0;
11        end loop;
12      end return;
13    end To_Angles;
14
15 end Angles;

```

Listing 13: rotation-tests.ads

```

1 package Rotation.Tests is
2
3   procedure Test_Rotation (N : Positive);
4
5   procedure Test_Angles (N : Positive);
6
7 end Rotation.Tests;

```

Listing 14: rotation-tests.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO; use Ada.Numerics;
3 with Ada.Numerics;          use Ada.Numerics;
4
5 with Angles;                use Angles;
6
7 package body Rotation.Tests is
8
9   package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
10  package F_IO is new Ada.Text_IO.Float_IO (Float);
11
12  --
13  -- Adapt value due to floating-point inaccuracies
14  --
15
16  function Adapt (C : Complex) return Complex is
17    function Check_Zero (F : Float) return Float is
18      (if F <= 0.0 and F >= -0.01 then 0.0 else F);
19
20 begin
21   return C_Out : Complex := C do
22     C_Out.Re := Check_Zero (C_Out.Re);
23     C_Out.Im := Check_Zero (C_Out.Im);
24   end return;
25 end Adapt;
26
27 function Adapt (A : Angle) return Angle is
28   (if A <= -179.99 and A >= -180.01 then 180.0 else A);
29
30 procedure Test_Rotation (N : Positive) is
31   C : constant Complex_Points := Rotation (N);
32
33 begin
34   Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
35   for V of C loop
36     Put ("Point: ");
37     C_IO.Put (Adapt (V), 0, 1, 0);
38     New_Line;
39   end loop;
40 end Test_Rotation;
41
42 procedure Test_Angles (N : Positive) is
43   C : constant Complex_Points := Rotation (N);
44   A : constant Angles.Angles := To_Angles (C);
45
46 begin
47   Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
48   for V of A loop
49     Put ("Angle: ");
50     F_IO.Put (Adapt (V), 0, 2, 0);
51     Put_Line (" degrees");
52   end loop;

```

(continues on next page)

(continued from previous page)

```

50    end Test_Angles;
51
52 end Rotation.Tests;
```

Listing 15: main.adb

```

1  with Ada.Command_Line;          use Ada.Command_Line;
2  with Ada.Text_IO;              use Ada.Text_IO;
3
4  with Rotation.Tests;          use Rotation.Tests;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Rotation_Chk,
9           Angles_Chk);
10
11     procedure Check (TC : Test_Case_Index; N : Positive) is
12     begin
13         case TC is
14             when Rotation_Chk =>
15                 Test_Rotation (N);
16             when Angles_Chk =>
17                 Test_Angles (N);
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 2 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 2 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
30 end Main;
```


SOLUTIONS

104.1 Imperative Language

104.1.1 Hello World

Listing 1: main.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Main is
4 begin
5     Put_Line ("Hello World!");
6 end Main;
```

104.1.2 Greetings

Listing 2: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 procedure Main is
5
6     procedure Greet (Name : String) is
7     begin
8         Put_Line ("Hello " & Name & "!");
9     end Greet;
10
11 begin
12     if Argument_Count < 1 then
13         Put_Line ("ERROR: missing arguments! Exiting... ");
14         return;
15     elsif Argument_Count > 1 then
16         Put_Line ("Ignoring additional arguments... ");
17     end if;
18
19     Greet (Argument (1));
20 end Main;
```

104.1.3 Positive Or Negative

Listing 3: classify_number.ads

```
1 procedure Classify_Number (X : Integer);
```

Listing 4: classify_number.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5   if X > 0 then
6     Put_Line ("Positive");
7   elsif X < 0 then
8     Put_Line ("Negative");
9   else
10    Put_Line ("Zero");
11  end if;
12 end Classify_Number;
```

Listing 5: main.adb

```
1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Classify_Number;
5
6 procedure Main is
7   A : Integer;
8 begin
9   if Argument_Count < 1 then
10     Put_Line ("ERROR: missing arguments! Exiting...");  
11     return;  
12   elsif Argument_Count > 1 then  
13     Put_Line ("Ignoring additional arguments...");  
14   end if;  
15
16   A := Integer'Value (Argument (1));
17
18   Classify_Number (A);
19 end Main;
```

104.1.4 Numbers

Listing 6: display_numbers.ads

```
1 procedure Display_Numbers (A, B : Integer);
```

Listing 7: display_numbers.adb

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Numbers (A, B : Integer) is
4   X, Y : Integer;
5 begin
6   if A <= B then
7     X := A;
```

(continues on next page)

(continued from previous page)

```

8     Y := B;
9  else
10    X := B;
11    Y := A;
12  end if;
13
14  for I in X .. Y loop
15    Put_Line (Integer'Image (I));
16  end loop;
17 end Display_Numbers;

```

Listing 8: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Numbers;
5
6 procedure Main is
7   A, B : Integer;
8 begin
9  if Argument_Count < 2 then
10    Put_Line ("ERROR: missing arguments! Exiting...");  

11    return;
12  elsif Argument_Count > 2 then
13    Put_Line ("Ignoring additional arguments...");  

14  end if;
15
16  A := Integer'Value (Argument (1));
17  B := Integer'Value (Argument (2));
18
19  Display_Numbers (A, B);
20 end Main;

```

104.2 Subprograms

104.2.1 Subtract Procedure

Listing 9: subtract.ads

```

1 procedure Subtract (A, B : Integer;
2                      Result : out Integer);

```

Listing 10: subtract.adb

```

1 procedure Subtract (A, B : Integer;
2                      Result : out Integer) is
3 begin
4  Result := A - B;
5 end Subtract;

```

Listing 11: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3

```

(continues on next page)

(continued from previous page)

```

4  with Subtract;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Sub_10_1_Chk,
9       Sub_10_100_Chk,
10      Sub_0_5_Chk,
11      Sub_0_Minus_5_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14    Result : Integer;
15  begin
16    case TC is
17      when Sub_10_1_Chk =>
18        Subtract (10, 1, Result);
19        Put_Line ("Result: " & Integer'Image (Result));
20      when Sub_10_100_Chk =>
21        Subtract (10, 100, Result);
22        Put_Line ("Result: " & Integer'Image (Result));
23      when Sub_0_5_Chk =>
24        Subtract (0, 5, Result);
25        Put_Line ("Result: " & Integer'Image (Result));
26      when Sub_0_Minus_5_Chk =>
27        Subtract (0, -5, Result);
28        Put_Line ("Result: " & Integer'Image (Result));
29    end case;
30  end Check;
31
32 begin
33  if Argument_Count < 1 then
34    Put_Line ("ERROR: missing arguments! Exiting... ");
35    return;
36  elsif Argument_Count > 1 then
37    Put_Line ("Ignoring additional arguments... ");
38  end if;
39
40  Check (Test_Case_Index'Value (Argument (1)));
41 end Main;

```

104.2.2 Subtract Function

Listing 12: subtract.ads

```

1  function Subtract (A, B : Integer) return Integer;

```

Listing 13: subtract.adb

```

1  function Subtract (A, B : Integer) return Integer is
2  begin
3    return A - B;
4  end Subtract;

```

Listing 14: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Subtract;

```

(continues on next page)

(continued from previous page)

```

5   procedure Main is
6     type Test_Case_Index is
7       (Sub_10_1_Chk,
8        Sub_10_100_Chk,
9        Sub_0_5_Chk,
10       Sub_0_Minus_5_Chk);
11
12   procedure Check (TC : Test_Case_Index) is
13     Result : Integer;
14   begin
15     case TC is
16       when Sub_10_1_Chk =>
17         Result := Subtract (10, 1);
18         Put_Line ("Result: " & Integer'Image (Result));
19       when Sub_10_100_Chk =>
20         Result := Subtract (10, 100);
21         Put_Line ("Result: " & Integer'Image (Result));
22       when Sub_0_5_Chk =>
23         Result := Subtract (0, 5);
24         Put_Line ("Result: " & Integer'Image (Result));
25       when Sub_0_Minus_5_Chk =>
26         Result := Subtract (0, -5);
27         Put_Line ("Result: " & Integer'Image (Result));
28     end case;
29   end Check;
30
31 begin
32   if Argument_Count < 1 then
33     Put_Line ("ERROR: missing arguments! Exiting... ");
34     return;
35   elsif Argument_Count > 1 then
36     Put_Line ("Ignoring additional arguments... ");
37   end if;
38
39   Check (Test_Case_Index'Value (Argument (1)));
40 end Main;
41

```

104.2.3 Equality function

Listing 15: is_equal.ads

```

1  function Is_Equal (A, B : Integer) return Boolean;

```

Listing 16: is_equal.adb

```

1  function Is_Equal (A, B : Integer) return Boolean is
2    begin
3      return A = B;
4    end Is_Equal;

```

Listing 17: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Is_Equal;
5

```

(continues on next page)

(continued from previous page)

```

6  procedure Main is
7    type Test_Case_Index is
8      (Equal_Chk,
9       Inequal_Chk);
10
11   procedure Check (TC : Test_Case_Index) is
12
13     procedure Display_Equal (A, B : Integer;
14                               Equal : Boolean) is
15       begin
16         Put (Integer'Image (A));
17         if Equal then
18           Put (" is equal to ");
19         else
20           Put (" isn't equal to ");
21         end if;
22         Put_Line (Integer'Image (B) & ".");
23     end Display_Equal;
24
25   Result : Boolean;
26 begin
27   case TC is
28     when Equal_Chk =>
29       for I in 0 .. 10 loop
30         Result := Is_Equal (I, I);
31         Display_Equal (I, I, Result);
32       end loop;
33     when Inequal_Chk =>
34       for I in 0 .. 10 loop
35         Result := Is_Equal (I, I - 1);
36         Display_Equal (I, I - 1, Result);
37       end loop;
38   end case;
39 end Check;
40
41 begin
42   if Argument_Count < 1 then
43     Put_Line ("ERROR: missing arguments! Exiting...");
44     return;
45   elsif Argument_Count > 1 then
46     Put_Line ("Ignoring additional arguments...");
47   end if;
48
49   Check (Test_Case_Index'Value (Argument (1)));
50 end Main;

```

104.2.4 States

Listing 18: display_state.ads

```

1  procedure Display_State (State : Integer);

```

Listing 19: display_state.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Display_State (State : Integer) is
4    begin

```

(continues on next page)

(continued from previous page)

```

5   case State is
6     when 0 =>
7       Put_Line ("Off");
8     when 1 =>
9       Put_Line ("On: Simple Processing");
10    when 2 =>
11      Put_Line ("On: Advanced Processing");
12    when others =>
13      null;
14  end case;
15 end Display_State;

```

Listing 20: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_State;
5
6 procedure Main is
7   State : Integer;
8 begin
9   if Argument_Count < 1 then
10     Put_Line ("ERROR: missing arguments! Exiting...");  

11     return;
12   elsif Argument_Count > 1 then
13     Put_Line ("Ignoring additional arguments...");  

14   end if;
15
16   State := Integer'Value (Argument (1));
17
18   Display_State (State);
19 end Main;

```

104.2.5 States #2

Listing 21: get_state.ads

```

1 function Get_State (State : Integer) return String;

```

Listing 22: get_state.adb

```

1 function Get_State (State : Integer) return String is
2 begin
3   return (case State is
4     when 0 => "Off",
5     when 1 => "On: Simple Processing",
6     when 2 => "On: Advanced Processing",
7     when others => "");  

8 end Get_State;

```

Listing 23: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Get_State;

```

(continues on next page)

(continued from previous page)

```

5  procedure Main is
6      State : Integer;
7  begin
8      if Argument_Count < 1 then
9          Put_Line ("ERROR: missing arguments! Exiting..."); 
10         return;
11     elsif Argument_Count > 1 then
12         Put_Line ("Ignoring additional arguments..."); 
13     end if;
14
15     State := Integer'Value (Argument (1));
16
17     Put_Line (Get_State (State));
18
19 end Main;

```

104.2.6 States #3

Listing 24: is_on.ads

```

1  function Is_On (State : Integer) return Boolean;

```

Listing 25: is_on.adb

```

1  function Is_On (State : Integer) return Boolean is
2  begin
3      return not (State = 0);
4  end Is_On;

```

Listing 26: display_on_off.ads

```

1  procedure Display_On_Off (State : Integer);

```

Listing 27: display_on_off.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Is_On;
3
4  procedure Display_On_Off (State : Integer) is
5  begin
6      Put_Line (if Is_On (State) then "On" else "Off");
7  end Display_On_Off;

```

Listing 28: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Display_On_Off;
5  with Is_On;
6
7  procedure Main is
8      State : Integer;
9  begin
10     if Argument_Count < 1 then
11         Put_Line ("ERROR: missing arguments! Exiting..."); 
12         return;

```

(continues on next page)

(continued from previous page)

```

13  elsif Argument_Count > 1 then
14      Put_Line ("Ignoring additional arguments..."); 
15  end if;
16
17  State := Integer'Value (Argument (1));
18
19  Display_On_Off (State);
20  Put_Line (Boolean'Image (Is_On (State)));
21 end Main;

```

104.2.7 States #4

Listing 29: set_next.ads

```
1 procedure Set_Next (State : in out Integer);
```

Listing 30: set_next.adb

```

1 procedure Set_Next (State : in out Integer) is
2 begin
3     State := (if State < 2 then State + 1 else 0);
4 end Set_Next;

```

Listing 31: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Set_Next;
5
6 procedure Main is
7     State : Integer;
8 begin
9     if Argument_Count < 1 then
10        Put_Line ("ERROR: missing arguments! Exiting..."); 
11        return;
12     elsif Argument_Count > 1 then
13        Put_Line ("Ignoring additional arguments..."); 
14     end if;
15
16     State := Integer'Value (Argument (1));
17
18     Set_Next (State);
19     Put_Line (Integer'Image (State));
20 end Main;

```

104.3 Modular Programming

104.3.1 Months

Listing 32: months.ads

```

1 package Months is
2
3     Jan : constant String := "January";
4     Feb : constant String := "February";
5     Mar : constant String := "March";
6     Apr : constant String := "April";
7     May : constant String := "May";
8     Jun : constant String := "June";
9     Jul : constant String := "July";
10    Aug : constant String := "August";
11    Sep : constant String := "September";
12    Oct : constant String := "October";
13    Nov : constant String := "November";
14    Dec : constant String := "December";
15
16    procedure Display_Months;
17
18 end Months;

```

Listing 33: months.adb

```

1 with Ada.Text_Io; use Ada.Text_Io;
2
3 package body Months is
4
5     procedure Display_Months is
6         begin
7             Put_Line ("Months:");
8             Put_Line ("- " & Jan);
9             Put_Line ("- " & Feb);
10            Put_Line ("- " & Mar);
11            Put_Line ("- " & Apr);
12            Put_Line ("- " & May);
13            Put_Line ("- " & Jun);
14            Put_Line ("- " & Jul);
15            Put_Line ("- " & Aug);
16            Put_Line ("- " & Sep);
17            Put_Line ("- " & Oct);
18            Put_Line ("- " & Nov);
19            Put_Line ("- " & Dec);
20        end Display_Months;
21
22 end Months;

```

Listing 34: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_Io;      use Ada.Text_Io;
3
4 with Months;          use Months;
5
6 procedure Main is
7
8     type Test_Case_Index is

```

(continues on next page)

(continued from previous page)

```

9      (Months_Chk);

10
11 procedure Check (TC : Test_Case_Index) is
12 begin
13   case TC is
14     when Months_Chk =>
15       Display_Months;
16   end case;
17 end Check;

18
19 begin
20   if Argument_Count < 1 then
21     Put_Line ("ERROR: missing arguments! Exiting...");  

22     return;
23   elsif Argument_Count > 1 then
24     Put_Line ("Ignoring additional arguments...");  

25   end if;
26
27   Check (Test_Case_Index'Value (Argument (1)));
28 end Main;

```

104.3.2 Operations

Listing 35: operations.ads

```

1 package Operations is
2
3   function Add (A, B : Integer) return Integer;
4
5   function Subtract (A, B : Integer) return Integer;
6
7   function Multiply (A, B : Integer) return Integer;
8
9   function Divide (A, B : Integer) return Integer;
10
11 end Operations;

```

Listing 36: operations.adb

```

1 package body Operations is
2
3   function Add (A, B : Integer) return Integer is
4 begin
5   return A + B;
6 end Add;
7
8   function Subtract (A, B : Integer) return Integer is
9 begin
10  return A - B;
11 end Subtract;
12
13   function Multiply (A, B : Integer) return Integer is
14 begin
15  return A * B;
16 end Multiply;
17
18   function Divide (A, B : Integer) return Integer is
19 begin

```

(continues on next page)

(continued from previous page)

```

20    return A / B;
21  end Divide;
22
23 end Operations;
```

Listing 37: operations-test.ads

```

1 package Operations.Test is
2
3   procedure Display (A, B : Integer);
4
5 end Operations.Test;
```

Listing 38: operations-test.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Operations.Test is
4
5   procedure Display (A, B : Integer) is
6     A_Str : constant String := Integer'Image (A);
7     B_Str : constant String := Integer'Image (B);
8   begin
9     Put_Line ("Operations:");
10    Put_Line (A_Str & " + " & B_Str & " = "
11              & Integer'Image (Add (A, B))
12              & ",");
13    Put_Line (A_Str & " - " & B_Str & " = "
14              & Integer'Image (Subtract (A, B))
15              & ",");
16    Put_Line (A_Str & " * " & B_Str & " = "
17              & Integer'Image (Multiply (A, B))
18              & ",");
19    Put_Line (A_Str & " / " & B_Str & " = "
20              & Integer'Image (Divide (A, B))
21              & ",");
22   end Display;
23
24 end Operations.Test;
```

Listing 39: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Operations;
5 with Operations.Test; use Operations.Test;
6
7 procedure Main is
8
9   type Test_Case_Index is
10     (Operations_Chk,
11      Operations_Display_Chk);
12
13   procedure Check (TC : Test_Case_Index) is
14   begin
15     case TC is
16       when Operations_Chk =>
17         Put_Line ("Add (100, 2) = "
18                   & Integer'Image (Operations.Add (100, 2)));

```

(continues on next page)

(continued from previous page)

```

19      Put_Line ("Subtract (100, 2) = "
20          & Integer'Image (Operations.Subtract (100, 2)));
21      Put_Line ("Multiply (100, 2) = "
22          & Integer'Image (Operations.Multiply (100, 2)));
23      Put_Line ("Divide (100, 2) = "
24          & Integer'Image (Operations.Divide (100, 2)));
25  when Operations_Display_Chk =>
26      Display (10, 5);
27      Display (1, 2);
28  end case;
29 end Check;
30
31 begin
32  if Argument_Count < 1 then
33      Put_Line ("ERROR: missing arguments! Exiting... ");
34      return;
35  elsif Argument_Count > 1 then
36      Put_Line ("Ignoring additional arguments... ");
37  end if;
38
39  Check (Test_Case_Index'Value (Argument (1)));
40 end Main;

```

104.4 Strongly typed language

104.4.1 Colors

Listing 40: color_types.ads

```

1 package Color_Types is
2
3     type HTML_Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     function To_Integer (C : HTML_Color) return Integer;
17
18     type Basic_HTML_Color is
19         (Red,
20          Green,
21          Blue);
22
23     function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25 end Color_Types;

```

Listing 41: color_types.adb

```

1 package body Color_Types is
2
3     function To_Integer (C : HTML_Color) return Integer is
4 begin
5     case C is
6         when Salmon      => return 16#FA8072#;
7         when Firebrick   => return 16#B22222#;
8         when Red         => return 16#FF0000#;
9         when Darkred    => return 16#8B0000#;
10        when Lime        => return 16#00FF00#;
11        when Forestgreen => return 16#228B22#;
12        when Green       => return 16#008000#;
13        when Darkgreen   => return 16#006400#;
14        when Blue        => return 16#0000FF#;
15        when Mediumblue  => return 16#0000CD#;
16        when Darkblue    => return 16#00008B#;
17    end case;
18
19 end To_Integer;
20
21 function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
22 begin
23     case C is
24         when Red      => return Red;
25         when Green    => return Green;
26         when Blue    => return Blue;
27     end case;
28 end To_HTML_Color;
29
30 end Color_Types;

```

Listing 42: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with Ada.Integer_Text_IO;
4
5 with Color_Types;      use Color_Types;
6
7 procedure Main is
8     type Test_Case_Index is
9         (HTML_Color_Range,
10          HTML_Color_To_Integer,
11          Basic_HTML_Color_To_HTML_Color);
12
13 procedure Check (TC : Test_Case_Index) is
14 begin
15     case TC is
16         when HTML_Color_Range =>
17             for I in HTML_Color'Range loop
18                 Put_Line (HTML_Color'Image (I));
19             end loop;
20         when HTML_Color_To_Integer =>
21             for I in HTML_Color'Range loop
22                 Ada.Integer_Text_IO.Put (Item  => To_Integer (I),
23                                         Width => 1,
24                                         Base   => 16);
25                 New_Line;
26             end loop;

```

(continues on next page)

(continued from previous page)

```

27      when Basic_HTML_Color_To_HTML_Color =>
28          for I in Basic_HTML_Color'Range loop
29              Put_Line (HTML_Color'Image (To_HTML_Color (I)));
30          end loop;
31      end case;
32  end Check;

33
34 begin
35     if Argument_Count < 1 then
36         Put_Line ("ERROR: missing arguments! Exiting...");  

37         return;
38     elsif Argument_Count > 1 then
39         Put_Line ("Ignoring additional arguments...");  

40     end if;
41
42     Check (Test_Case_Index'Value (Argument (1)));
43 end Main;

```

104.4.2 Integers

Listing 43: int_types.ads

```

1 package Int_Types is
2
3     type I_100 is range 0 .. 100;
4
5     type U_100 is mod 101;
6
7     function To_I_100 (V : U_100) return I_100;
8
9     function To_U_100 (V : I_100) return U_100;
10
11    type D_50 is new I_100 range 10 .. 50;
12
13    subtype S_50 is I_100 range 10 .. 50;
14
15    function To_D_50 (V : I_100) return D_50;
16
17    function To_S_50 (V : I_100) return S_50;
18
19    function To_I_100 (V : D_50) return I_100;
20
21 end Int_Types;

```

Listing 44: int_types.adb

```

1 package body Int_Types is
2
3     function To_I_100 (V : U_100) return I_100 is
4     begin
5         return I_100 (V);
6     end To_I_100;
7
8     function To_U_100 (V : I_100) return U_100 is
9     begin
10        return U_100 (V);
11    end To_U_100;
12

```

(continues on next page)

(continued from previous page)

```

13  function To_D_50 (V : I_100) return D_50 is
14    Min : constant I_100 := I_100 (D_50'First);
15    Max : constant I_100 := I_100 (D_50'Last);
16 begin
17   if V > Max then
18     return D_50'Last;
19   elsif V < Min then
20     return D_50'First;
21   else
22     return D_50 (V);
23   end if;
24 end To_D_50;
25
26 function To_S_50 (V : I_100) return S_50 is
27 begin
28   if V > S_50'Last then
29     return S_50'Last;
30   elsif V < S_50'First then
31     return S_50'First;
32   else
33     return V;
34   end if;
35 end To_S_50;
36
37 function To_I_100 (V : D_50) return I_100 is
38 begin
39   return I_100 (V);
40 end To_I_100;
41
42 end Int_Types;

```

Listing 45: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3
4  with Int_Types;        use Int_Types;
5
6  procedure Main is
7    package I_100_IO is new Ada.Text_IO.Integer_IO (I_100);
8    package U_100_IO is new Ada.Text_IO.Modular_IO (U_100);
9    package D_50_IO  is new Ada.Text_IO.Integer_IO (D_50);
10
11  use I_100_IO;
12  use U_100_IO;
13  use D_50_IO;
14
15  type Test_Case_Index is
16    (I_100_Range,
17     U_100_Range,
18     U_100_Wraparound,
19     U_100_To_I_100,
20     I_100_To_U_100,
21     D_50_Range,
22     S_50_Range,
23     I_100_To_D_50,
24     I_100_To_S_50,
25     D_50_To_I_100,
26     S_50_To_I_100);
27
28  procedure Check (TC : Test_Case_Index) is

```

(continues on next page)

(continued from previous page)

```

29 begin
30   I_100_IO.Default_Width := 1;
31   U_100_IO.Default_Width := 1;
32   D_50_IO.Default_Width := 1;
33
34   case TC is
35     when I_100_Range =>
36       Put(I_100'First);
37       New_Line;
38       Put(I_100'Last);
39       New_Line;
40     when U_100_Range =>
41       Put(U_100'First);
42       New_Line;
43       Put(U_100'Last);
44       New_Line;
45     when U_100_Wraparound =>
46       Put(U_100'First - 1);
47       New_Line;
48       Put(U_100'Last + 1);
49       New_Line;
50     when U_100_To_I_100 =>
51       for I in U_100'Range loop
52         I_100_IO.Put(To_I_100(I));
53         New_Line;
54       end loop;
55     when I_100_To_U_100 =>
56       for I in I_100'Range loop
57         Put(To_U_100(I));
58         New_Line;
59       end loop;
60     when D_50_Range =>
61       Put(D_50'First);
62       New_Line;
63       Put(D_50'Last);
64       New_Line;
65     when S_50_Range =>
66       Put(S_50'First);
67       New_Line;
68       Put(S_50'Last);
69       New_Line;
70     when I_100_To_D_50 =>
71       for I in I_100'Range loop
72         Put(To_D_50(I));
73         New_Line;
74       end loop;
75     when I_100_To_S_50 =>
76       for I in I_100'Range loop
77         Put(To_S_50(I));
78         New_Line;
79       end loop;
80     when D_50_To_I_100 =>
81       for I in D_50'Range loop
82         Put(To_I_100(I));
83         New_Line;
84       end loop;
85     when S_50_To_I_100 =>
86       for I in S_50'Range loop
87         Put(I);
88         New_Line;
89       end loop;

```

(continues on next page)

(continued from previous page)

```

90      end case;
91  end Check;

92
93 begin
94  if Argument_Count < 1 then
95    Put_Line ("ERROR: missing arguments! Exiting...");  

96    return;
97  elsif Argument_Count > 1 then
98    Put_Line ("Ignoring additional arguments...");  

99  end if;
100
101  Check (Test_Case_Index'Value (Argument (1)));
102 end Main;

```

104.4.3 Temperatures

Listing 46: temperature_types.ads

```

1 package Temperature_Types is
2
3   type Celsius is digits 6 range -273.15 .. 5504.85;
4
5   type Int_Celsius is range -273 .. 5505;
6
7   function To_Celsius (T : Int_Celsius) return Celsius;
8
9   function To_Int_Celsius (T : Celsius) return Int_Celsius;
10
11  type Kelvin is digits 6 range 0.0 .. 5778.00;
12
13  function To_Celsius (T : Kelvin) return Celsius;
14
15  function To_Kelvin (T : Celsius) return Kelvin;
16
17 end Temperature_Types;

```

Listing 47: temperature_types.adb

```

1 package body Temperature_Types is
2
3   function To_Celsius (T : Int_Celsius) return Celsius is
4     Min : constant Float := Float (Celsius'First);
5     Max : constant Float := Float (Celsius'Last);
6
7     F : constant Float := Float (T);
8
9   begin
10    if F > Max then
11      return Celsius (Max);
12    elsif F < Min then
13      return Celsius (Min);
14    else
15      return Celsius (F);
16    end if;
17  end To_Celsius;
18
19  function To_Int_Celsius (T : Celsius) return Int_Celsius is
20  begin
21    return Int_Celsius (T);

```

(continues on next page)

(continued from previous page)

```

21 end To_Int_Celsius;

22 function To_Celsius (T : Kelvin) return Celsius is
23   F : constant Float := Float (T);
24 begin
25   return Celsius (F - 273.15);
26 end To_Celsius;

27 function To_Kelvin (T : Celsius) return Kelvin is
28   F : constant Float := Float (T);
29 begin
30   return Kelvin (F + 273.15);
31 end To_Kelvin;

32
33
34
35 end Temperature_Types;

```

Listing 48: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_Io;       use Ada.Text_Io;
3
4 with Temperature_Types; use Temperature_Types;
5
6 procedure Main is
7   package Celsius_Io      is new Ada.Text_Io.Float_Io (Celsius);
8   package Kelvin_Io       is new Ada.Text_Io.Float_Io (Kelvin);
9   package Int_Celsius_Io is new Ada.Text_Io.Integer_Io (Int_Celsius);
10
11 use Celsius_Io;
12 use Kelvin_Io;
13 use Int_Celsius_Io;
14
15 type Test_Case_Index is
16   (Celsius_Range,
17    Celsius_To_Int_Celsius,
18    Int_Celsius_To_Celsius,
19    Kelvin_To_Celsius,
20    Celsius_To_Kelvin);
21
22 procedure Check (TC : Test_Case_Index) is
23 begin
24   Celsius_Io.Default_Fore := 1;
25   Kelvin_Io.Default_Fore := 1;
26   Int_Celsius_Io.Default_Width := 1;
27
28   case TC is
29     when Celsius_Range =>
30       Put (Celsius'First);
31       New_Line;
32       Put (Celsius'Last);
33       New_Line;
34     when Celsius_To_Int_Celsius =>
35       Put (To_Int_Celsius (Celsius'First));
36       New_Line;
37       Put (To_Int_Celsius (0.0));
38       New_Line;
39       Put (To_Int_Celsius (Celsius'Last));
40       New_Line;
41     when Int_Celsius_To_Celsius =>
42       Put (To_Celsius (Int_Celsius'First));
43       New_Line;

```

(continues on next page)

(continued from previous page)

```

44      Put (To_Celsius (0));
45      New_Line;
46      Put (To_Celsius (Int_Celsius'Last));
47      New_Line;
48      when Kelvin_To_Celsius =>
49          Put (To_Celsius (Kelvin'First));
50          New_Line;
51          Put (To_Celsius (0));
52          New_Line;
53          Put (To_Celsius (Kelvin'Last));
54          New_Line;
55      when Celsius_To_Kelvin =>
56          Put (To_Kelvin (Celsius'First));
57          New_Line;
58          Put (To_Kelvin (Celsius'Last));
59          New_Line;
60      end case;
61  end Check;

62
63 begin
64  if Argument_Count < 1 then
65      Put_Line ("ERROR: missing arguments! Exiting..."); 
66      return;
67  elsif Argument_Count > 1 then
68      Put_Line ("Ignoring additional arguments..."); 
69  end if;
70
71  Check (Test_Case_Index'Value (Argument (1)));
72 end Main;

```

104.5 Records

104.5.1 Directions

Listing 49: directions.ads

```

1 package Directions is
2
3     type Angle_Mod is mod 360;
4
5     type Direction is
6         (North,
7          Northeast,
8          East,
9          Southeast,
10         South,
11         Southwest,
12         West,
13         Northwest);
14
15     function To_Direction (N: Angle_Mod) return Direction;
16
17     type Ext_Angle is record
18         Angle_Elem    : Angle_Mod;
19         Direction_Elem : Direction;
20     end record;
21

```

(continues on next page)

(continued from previous page)

```

22  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
23
24  procedure Display (N : Ext_Angle);
25
26 end Directions;

```

Listing 50: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5    procedure Display (N : Ext_Angle) is
6    begin
7      Put_Line ("Angle: "
8                & Angle_Mod'Image (N.Angle_Elem)
9                & " => "
10               & Direction'Image (N.Direction_Elem)
11               & ".");
12  end Display;
13
14  function To_Direction (N : Angle_Mod) return Direction is
15  begin
16    case N is
17      when 0          => return North;
18      when 1 .. 89   => return Northeast;
19      when 90        => return East;
20      when 91 .. 179 => return Southeast;
21      when 180       => return South;
22      when 181 .. 269 => return Southwest;
23      when 270       => return West;
24      when 271 .. 359 => return Northwest;
25    end case;
26  end To_Direction;
27
28  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29  begin
30    return (Angle_Elem      => N,
31            Direction_Elem => To_Direction (N));
32  end To_Ext_Angle;
33
34 end Directions;

```

Listing 51: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Directions;         use Directions;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Direction_Chk);
9
10 procedure Check (TC : Test_Case_Index) is
11 begin
12   case TC is
13     when Direction_Chk =>
14       Display (To_Ext_Angle (0));
15       Display (To_Ext_Angle (30));

```

(continues on next page)

(continued from previous page)

```

16    Display (To_Ext_Angle (45));
17    Display (To_Ext_Angle (90));
18    Display (To_Ext_Angle (91));
19    Display (To_Ext_Angle (120));
20    Display (To_Ext_Angle (180));
21    Display (To_Ext_Angle (250));
22    Display (To_Ext_Angle (270));
23  end case;
24 end Check;

25
26 begin
27  if Argument_Count < 1 then
28    Put_Line ("ERROR: missing arguments! Exiting...");  

29    return;
30  elsif Argument_Count > 1 then
31    Put_Line ("Ignoring additional arguments...");  

32  end if;
33
34  Check (Test_Case_Index'Value (Argument (1)));
35 end Main;

```

104.5.2 Colors

Listing 52: color_types.ads

```

1 package Color_Types is
2
3   type HTML_Color is
4     (Salmon,
5      Firebrick,
6      Red,
7      Darkred,
8      Lime,
9      Forestgreen,
10     Green,
11     Darkgreen,
12     Blue,
13     Mediumblue,
14     Darkblue);
15
16   function To_Integer (C : HTML_Color) return Integer;
17
18   type Basic_HTML_Color is
19     (Red,
20      Green,
21      Blue);
22
23   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color;
24
25   subtype Int_Color is Integer range 0 .. 255;
26
27   type RGB is record
28     Red   : Int_Color;
29     Green : Int_Color;
30     Blue  : Int_Color;
31   end record;
32
33   function To_RGB (C : HTML_Color) return RGB;
34

```

(continues on next page)

(continued from previous page)

```

35  function Image (C : RGB) return String;
36
37 end Color_Types;
```

Listing 53: color_types.adb

```

1  with Ada.Integer_Text_Io;
2
3 package body Color_Types is
4
5   function To_Integer (C : HTML_Color) return Integer is
6   begin
7     case C is
8       when Salmon      => return 16#FA8072#;
9       when Firebrick   => return 16#B22222#;
10      when Red         => return 16#FF0000#;
11      when Darkred    => return 16#8B0000#;
12      when Lime        => return 16#00FF00#;
13      when Forestgreen => return 16#228B22#;
14      when Green       => return 16#008000#;
15      when Darkgreen   => return 16#006400#;
16      when Blue        => return 16#0000FF#;
17      when Mediumblue  => return 16#0000CD#;
18      when Darkblue    => return 16#00008B#;
19   end case;
20
21   end To_Integer;
22
23   function To_HTML_Color (C : Basic_HTML_Color) return HTML_Color is
24   begin
25     case C is
26       when Red      => return Red;
27       when Green    => return Green;
28       when Blue    => return Blue;
29     end case;
30   end To_HTML_Color;
31
32   function To_RGB (C : HTML_Color) return RGB is
33   begin
34     case C is
35       when Salmon      => return (16#FA#, 16#80#, 16#72#);
36       when Firebrick   => return (16#B2#, 16#22#, 16#22#);
37       when Red         => return (16#FF#, 16#00#, 16#00#);
38       when Darkred    => return (16#8B#, 16#00#, 16#00#);
39       when Lime        => return (16#00#, 16#FF#, 16#00#);
40       when Forestgreen => return (16#22#, 16#8B#, 16#22#);
41       when Green       => return (16#00#, 16#80#, 16#00#);
42       when Darkgreen   => return (16#00#, 16#64#, 16#00#);
43       when Blue        => return (16#00#, 16#00#, 16#FF#);
44       when Mediumblue  => return (16#00#, 16#00#, 16#CD#);
45       when Darkblue    => return (16#00#, 16#00#, 16#8B#);
46     end case;
47
48   end To_RGB;
49
50   function Image (C : RGB) return String is
51     subtype Str_Range is Integer range 1 .. 10;
52     SR : String (Str_Range);
53     SG : String (Str_Range);
54     SB : String (Str_Range);
55   begin
```

(continues on next page)

(continued from previous page)

```

56     Ada.Integer_Text_Io.Put (To    => SR,
57                             Item  => C.Red,
58                             Base   => 16);
59     Ada.Integer_Text_Io.Put (To    => SG,
60                             Item  => C.Green,
61                             Base   => 16);
62     Ada.Integer_Text_Io.Put (To    => SB,
63                             Item  => C.Blue,
64                             Base   => 16);
65     return ("(Red => " & SR
66             & ", Green => " & SG
67             & ", Blue => " & SB
68             & ")");
69 end Image;
70
71 end Color_Types;

```

Listing 54: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_Io;         use Ada.Text_Io;
3
4  with Color_Types;        use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_To_RGB);
9
10 procedure Check (TC : Test_Case_Index) is
11 begin
12     case TC is
13         when HTML_Color_To_RGB =>
14             for I in HTML_Color'Range loop
15                 Put_Line (HTML_Color'Image (I) & " => "
16                           & Image (To_RGB (I)) & ".");
17             end loop;
18     end case;
19 end Check;
20
21 begin
22     if Argument_Count < 1 then
23         Put_Line ("ERROR: missing arguments! Exiting... ");
24         return;
25     elsif Argument_Count > 1 then
26         Put_Line ("Ignoring additional arguments... ");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)));
30 end Main;

```

104.5.3 Inventory

Listing 55: inventory_pkg.ads

```

1 package Inventory_Pkg is
2
3     type Item_Name is
4         (Ballpoint_Pen, Oil_Based_Pen_Marker, Feather_Quill_Pen);
5
6     function To_String (I : Item_Name) return String;
7
8     type Item is record
9         Name      : Item_Name;
10        Quantity : Natural;
11        Price    : Float;
12    end record;
13
14    function Init (Name      : Item_Name;
15                  Quantity : Natural;
16                  Price    : Float) return Item;
17
18    procedure Add (Assets : in out Float;
19                   I       : Item);
20
21 end Inventory_Pkg;

```

Listing 56: inventory_pkg.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Inventory_Pkg is
4
5     function To_String (I : Item_Name) return String is
6     begin
7         case I is
8             when Ballpoint_Pen      => return "Ballpoint Pen";
9             when Oil_Based_Pen_Marker => return "Oil-based Pen Marker";
10            when Feather_Quill_Pen   => return "Feather Quill Pen";
11        end case;
12    end To_String;
13
14    function Init (Name      : Item_Name;
15                  Quantity : Natural;
16                  Price    : Float) return Item is
17    begin
18        Put_Line ("Item: " & To_String (Name) & ".");
19
20        return (Name      => Name,
21                Quantity => Quantity,
22                Price    => Price);
23    end Init;
24
25    procedure Add (Assets : in out Float;
26                   I       : Item) is
27    begin
28        Assets := Assets + Float (I.Quantity) * I.Price;
29    end Add;
30
31 end Inventory_Pkg;

```

Listing 57: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Inventory_Pkg;      use Inventory_Pkg;
5
6  procedure Main is
7    -- Remark: the following line is not relevant.
8    F   : array (1 .. 10) of Float := (others => 42.42);
9
10 type Test_Case_Index is
11   (Inventory_Chk);
12
13 procedure Display (Assets : Float) is
14   package F_IO is new Ada.Text_IO.Float_IO (Float);
15
16   use F_IO;
17 begin
18   Put ("Assets: $");
19   Put (Assets, 1, 2, 0);
20   Put (".");
21   New_Line;
22 end Display;
23
24 procedure Check (TC : Test_Case_Index) is
25   I      : Item;
26   Assets : Float := 0.0;
27
28   -- Please ignore the following three lines!
29   pragma Warnings (Off, "default initialization");
30   for Assets'Address use F'Address;
31   pragma Warnings (On, "default initialization");
32 begin
33   case TC is
34     when Inventory_Chk =>
35       I := Init (Ballpoint_Pen,          185,  0.15);
36       Add (Assets, I);
37       Display (Assets);
38
39       I := Init (Oil_Based_Pen_Marker, 100,  9.0);
40       Add (Assets, I);
41       Display (Assets);
42
43       I := Init (Feather_Quill_Pen,     2, 40.0);
44       Add (Assets, I);
45       Display (Assets);
46   end case;
47 end Check;
48
49 begin
50   if Argument_Count < 1 then
51     Put_Line ("ERROR: missing arguments! Exiting...");
52     return;
53   elsif Argument_Count > 1 then
54     Put_Line ("Ignoring additional arguments...");
55   end if;
56
57   Check (Test_Case_Index'Value (Argument (1)));
58 end Main;

```

104.6 Arrays

104.6.1 Constrained Array

Listing 58: constrained_arrays.ads

```

1 package Constrained_Arrays is
2
3     type My_Index is range 1 .. 10;
4
5     type My_Array is array (My_Index) of Integer;
6
7     function Init return My_Array;
8
9     procedure Double (A : in out My_Array);
10
11    function First_Elem (A : My_Array) return Integer;
12
13    function Last_Elem (A : My_Array) return Integer;
14
15    function Length (A : My_Array) return Integer;
16
17    A : My_Array := (1, 2, others => 42);
18
19 end Constrained_Arrays;

```

Listing 59: constrained_arrays.adb

```

1 package body Constrained_Arrays is
2
3     function Init return My_Array is
4         A : My_Array;
5     begin
6         for I in My_Array'Range loop
7             A (I) := Integer (I);
8         end loop;
9
10        return A;
11    end Init;
12
13    procedure Double (A : in out My_Array) is
14    begin
15        for I in A'Range loop
16            A (I) := A (I) * 2;
17        end loop;
18    end Double;
19
20    function First_Elem (A : My_Array) return Integer is
21    begin
22        return A (A'First);
23    end First_Elem;
24
25    function Last_Elem (A : My_Array) return Integer is
26    begin
27        return A (A'Last);
28    end Last_Elem;
29
30    function Length (A : My_Array) return Integer is
31    begin
32        return A'Length;

```

(continues on next page)

(continued from previous page)

```

33  end Length;
34
35 end Constrained_Arrays;
```

Listing 60: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Constrained_Arrays; use Constrained_Arrays;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Range_Chk,
9       Array_Range_Chk,
10      A_Obj_Chk,
11      Init_Chk,
12      Double_Chk,
13      First_Elem_Chk,
14      Last_Elem_Chk,
15      Length_Chk);
16
17  procedure Check (TC : Test_Case_Index) is
18    AA : My_Array;
19
20    procedure Display (A : My_Array) is
21    begin
22      for I in A'Range loop
23        Put_Line (Integer'Image (A (I)));
24      end loop;
25    end Display;
26
27    procedure Local_Init (A : in out My_Array) is
28    begin
29      A := (100, 90, 80, 10, 20, 30, 40, 60, 50, 70);
30    end Local_Init;
31
32 begin
33   case TC is
34     when Range_Chk =>
35       for I in My_Index loop
36         Put_Line (My_Index'Image (I));
37       end loop;
38     when Array_Range_Chk =>
39       for I in My_Array'Range loop
40         Put_Line (My_Index'Image (I));
41       end loop;
42     when A_Obj_Chk =>
43       Display (A);
44     when Init_Chk =>
45       AA := Init;
46       Display (AA);
47     when Double_Chk =>
48       Local_Init (AA);
49       Double (AA);
50       Display (AA);
51     when First_Elem_Chk =>
52       Local_Init (AA);
53       Put_Line (Integer'Image (First_Elem (AA)));
54     when Last_Elem_Chk =>
55       Local_Init (AA);
56       Put_Line (Integer'Image (Last_Elem (AA)));
```

(continues on next page)

(continued from previous page)

```

56      when Length_Chk =>
57          Put_Line (Integer'Image (Length (AA)));
58      end case;
59  end Check;

60
61 begin
62  if Argument_Count < 1 then
63      Put_Line ("ERROR: missing arguments! Exiting... ");
64      return;
65  elsif Argument_Count > 1 then
66      Put_Line ("Ignoring additional arguments... ");
67  end if;
68
69  Check (Test_Case_Index'Value (Argument (1)));
70 end Main;

```

104.6.2 Colors: Lookup-Table

Listing 61: color_types.ads

```

1 package Color_Types is
2
3     type HTML_Color is
4         (Salmon,
5          Firebrick,
6          Red,
7          Darkred,
8          Lime,
9          Forestgreen,
10         Green,
11         Darkgreen,
12         Blue,
13         Mediumblue,
14         Darkblue);
15
16     subtype Int_Color is Integer range 0 .. 255;
17
18     type RGB is record
19         Red   : Int_Color;
20         Green : Int_Color;
21         Blue  : Int_Color;
22     end record;
23
24     function To_RGB (C : HTML_Color) return RGB;
25
26     function Image (C : RGB) return String;
27
28     type HTML_Color_RGB is array (HTML_Color) of RGB;
29
30     To_RGB_Lookup_Table : constant HTML_Color_RGB
31         := (Salmon      => (16#FA#, 16#80#, 16#72#),
32             Firebrick   => (16#B2#, 16#22#, 16#22#),
33             Red         => (16#FF#, 16#00#, 16#00#),
34             Darkred    => (16#8B#, 16#00#, 16#00#),
35             Lime        => (16#00#, 16#FF#, 16#00#),
36             Forestgreen => (16#22#, 16#8B#, 16#22#),
37             Green       => (16#00#, 16#80#, 16#00#),
38             Darkgreen   => (16#00#, 16#64#, 16#00#),
39             Blue        => (16#00#, 16#00#, 16#FF#),

```

(continues on next page)

(continued from previous page)

```

40      Mediumblue  => (16#00#, 16#00#, 16#CD#),
41      Darkblue    => (16#00#, 16#00#, 16#8B#));
42
43 end Color_Types;

```

Listing 62: color_types.adb

```

1  with Ada.Integer_Text_IO;
2  package body Color_Types is
3
4      function To_RGB (C : HTML_Color) return RGB is
5      begin
6          return To_RGB_Lookup_Table (C);
7      end To_RGB;
8
9      function Image (C : RGB) return String is
10         subtype Str_Range is Integer range 1 .. 10;
11         SR : String (Str_Range);
12         SG : String (Str_Range);
13         SB : String (Str_Range);
14
15         Ada.Integer_Text_IO.Put (To      => SR,
16                               Item   => C.Red,
17                               Base   => 16);
18         Ada.Integer_Text_IO.Put (To      => SG,
19                               Item   => C.Green,
20                               Base   => 16);
21         Ada.Integer_Text_IO.Put (To      => SB,
22                               Item   => C.Blue,
23                               Base   => 16);
24
25         return ("(Red => " & SR
26                 & ", Green => " & SG
27                 & ", Blue => " & SB
28                 &" )");
29
30 end Color_Types;

```

Listing 63: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Color_Types;          use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Color_Table_Chk,
9           HTML_Color_To_Integer_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12  begin
13      case TC is
14          when Color_Table_Chk =>
15              Put_Line ("Size of HTML_Color_RGB: "
16                        & Integer'Image (HTML_Color_RGB'Length));
17              Put_Line ("Firebrick: "
18                        & Image (To_RGB_Lookup_Table (Firebrick)));
19          when HTML_Color_To_Integer_Chk =>
20              for I in HTML_Color'Range loop

```

(continues on next page)

(continued from previous page)

```

21      Put_Line (HTML_Color'Image (I) & " => "
22                  & Image (To_RGB (I)) & ".");
23      end loop;
24  end case;
25 end Check;

26
27 begin
28  if Argument_Count < 1 then
29      Put_Line ("ERROR: missing arguments! Exiting...");  

30      return;
31  elsif Argument_Count > 1 then
32      Put_Line ("Ignoring additional arguments...");  

33  end if;
34
35  Check (Test_Case_Index'Value (Argument (1)));
36 end Main;

```

104.6.3 Unconstrained Array

Listing 64: unconstrained_arrays.ads

```

1 package Unconstrained_Arrays is
2
3     type My_Array is array (Positive range <>) of Integer;
4
5     procedure Init (A : in out My_Array);
6
7     function Init (I, L : Positive) return My_Array;
8
9     procedure Double (A : in out My_Array);
10
11    function Diff_Prev_Elem (A : My_Array) return My_Array;
12
13 end Unconstrained_Arrays;

```

Listing 65: unconstrained_arrays.adb

```

1 package body Unconstrained_Arrays is
2
3     procedure Init (A : in out My_Array) is
4         Y : Natural := A'Last;
5     begin
6         for I in A'Range loop
7             A (I) := Y;
8             Y := Y - 1;
9         end loop;
10    end Init;
11
12    function Init (I, L : Positive) return My_Array is
13        A : My_Array (I .. I + L - 1);
14    begin
15        Init (A);
16        return A;
17    end Init;
18
19    procedure Double (A : in out My_Array) is
20    begin
21        for I in A'Range loop

```

(continues on next page)

(continued from previous page)

```

22      A (I) := A (I) * 2;
23   end loop;
24 end Double;

25
26 function Diff_Prev_Elem (A : My_Array) return My_Array is
27   A_Out : My_Array (A'Range);
28 begin
29   A_Out (A'First) := 0;
30   for I in A'First + 1 .. A'Last loop
31     A_Out (I) := A (I) - A (I - 1);
32   end loop;
33
34   return A_Out;
35 end Diff_Prev_Elem;
36
37 end Unconstrained_Arrays;

```

Listing 66: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Unconstrained_Arrays; use Unconstrained_Arrays;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Init_Chk,
9       Init_Proc_Chk,
10      Double_Chk,
11      Diff_Prev_Chk,
12      Diff_Prev_Single_Chk);
13
14  procedure Check (TC : Test_Case_Index) is
15    AA : My_Array (1 .. 5);
16    AB : My_Array (5 .. 9);
17
18  procedure Display (A : My_Array) is
19  begin
20    for I in A'Range loop
21      Put_Line (Integer'Image (A (I)));
22    end loop;
23  end Display;
24
25  procedure Local_Init (A : in out My_Array) is
26  begin
27    A := (1, 2, 5, 10, -10);
28  end Local_Init;
29
30  begin
31    case TC is
32    when Init_Chk =>
33      AA := Init (AA'First, AA'Length);
34      AB := Init (AB'First, AB'Length);
35      Display (AA);
36      Display (AB);
37    when Init_Proc_Chk =>
38      Init (AA);
39      Init (AB);
40      Display (AA);
41      Display (AB);
42    when Double_Chk =>

```

(continues on next page)

(continued from previous page)

```

43      Local_Init (AB);
44      Double (AB);
45      Display (AB);
46  when Diff_Prev_Chk =>
47      Local_Init (AB);
48      AB := Diff_Prev_Elem (AB);
49      Display (AB);
50  when Diff_Prev_Single_Chk =>
51      declare
52          A1 : My_Array (1 .. 1) := (1 => 42);
53      begin
54          A1 := Diff_Prev_Elem (A1);
55          Display (A1);
56      end;
57  end case;
58 end Check;

59
60 begin
61  if Argument_Count < 1 then
62      Put_Line ("ERROR: missing arguments! Exiting...");  

63      return;
64  elsif Argument_Count > 1 then
65      Put_Line ("Ignoring additional arguments...");  

66  end if;
67
68  Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

104.6.4 Product info

Listing 67: product_info_pkg.ads

```

1 package Product_Info_Pkg is
2
3     subtype Quantity is Natural;
4
5     subtype Currency is Float;
6
7     type Product_Info is record
8         Units : Quantity;
9         Price : Currency;
10    end record;
11
12    type Product_Infos is array (Positive range <>) of Product_Info;
13
14    type Currency_Array is array (Positive range <>) of Currency;
15
16    procedure Total (P   : Product_Infos;
17                     Tot : out Currency_Array);
18
19    function Total (P : Product_Infos) return Currency_Array;
20
21    function Total (P : Product_Infos) return Currency;
22
23 end Product_Info_Pkg;

```

Listing 68: product_info_pkg.adb

```

1 package body Product_Info_Pkg is
2
3     -- Get total for single product
4     function Total (P : Product_Info) return Currency is
5         (Currency (P.Units) * P.Price);
6
7     procedure Total (P   : Product_Infos;
8                      Tot : out Currency_Array) is
9     begin
10        for I in P'Range loop
11            Tot (I) := Total (P (I));
12        end loop;
13    end Total;
14
15    function Total (P : Product_Infos) return Currency_Array
16    is
17        Tot : Currency_Array (P'Range);
18    begin
19        Total (P, Tot);
20        return Tot;
21    end Total;
22
23    function Total (P : Product_Infos) return Currency
24    is
25        Tot : Currency := 0.0;
26    begin
27        for I in P'Range loop
28            Tot := Tot + Total (P (I));
29        end loop;
30        return Tot;
31    end Total;
32
33 end Product_Info_Pkg;

```

Listing 69: main.adb

```

1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;          use Ada.Text_IO;
3
4 with Product_Info_Pkg;    use Product_Info_Pkg;
5
6 procedure Main is
7     package Currency_IO is new Ada.Text_IO.Float_IO (Currency);
8
9     type Test_Case_Index is
10        (Total_Func_Chk,
11         Total_Proc_Chk,
12         Total_Value_Chk);
13
14     procedure Check (TC : Test_Case_Index) is
15         subtype Test_Range is Positive range 1 .. 5;
16
17         P      : Product_Infos (Test_Range);
18         Tots  : Currency_Array (Test_Range);
19         Tot   : Currency;
20
21         procedure Display (Tots : Currency_Array) is
22         begin
23             for I in Tots'Range loop

```

(continues on next page)

(continued from previous page)

```

24     Currency_I0.Put (Tots (I));
25     New_Line;
26   end loop;
27 end Display;

28
29 procedure Local_Init (P : in out Product_Infos) is
30 begin
31   P := ((1,    0.5),
32         (2,   10.0),
33         (5,   40.0),
34         (10,  10.0),
35         (10,  20.0));
36 end Local_Init;

37
38 begin
39   Currency_I0.Default_Fore := 1;
40   Currency_I0.Default_Aft  := 2;
41   Currency_I0.Default_Exp   := 0;

42
43   case TC is
44     when Total_Func_Chk =>
45       Local_Init (P);
46       Tots := Total (P);
47       Display (Tots);
48     when Total_Proc_Chk =>
49       Local_Init (P);
50       Total (P, Tots);
51       Display (Tots);
52     when Total_Value_Chk =>
53       Local_Init (P);
54       Tot := Total (P);
55       Currency_I0.Put (Tot);
56       New_Line;
57   end case;
58 end Check;

59
60 begin
61   if Argument_Count < 1 then
62     Put_Line ("ERROR: missing arguments! Exiting...");
63     return;
64   elsif Argument_Count > 1 then
65     Put_Line ("Ignoring additional arguments...");
66   end if;

67
68   Check (Test_Case_Index'Value (Argument (1)));
69 end Main;

```

104.6.5 String_10

Listing 70: strings_10.ads

```

1 package Strings_10 is
2
3   subtype String_10 is String (1 .. 10);
4
5   -- Using "type String_10 is..." is possible, too.
6
7   function To_String_10 (S : String) return String_10;

```

(continues on next page)

(continued from previous page)

```
8
9 end Strings_10;
```

Listing 71: strings_10.adb

```
1 package body Strings_10 is
2
3     function To_String_10 (S : String) return String_10 is
4         S_Out : String_10;
5     begin
6         for I in String_10'First .. Integer'Min (String_10'Last, S'Last) loop
7             S_Out (I) := S (I);
8         end loop;
9
10        for I in Integer'Min (String_10'Last + 1, S'Last + 1) .. String_10'Last loop
11            S_Out (I) := ' ';
12        end loop;
13
14        return S_Out;
15    end To_String_10;
16
17 end Strings_10;
```

Listing 72: main.adb

```
1 with Ada.Command_Line;      use Ada.Command_Line;
2 with Ada.Text_IO;           use Ada.Text_IO;
3
4 with Strings_10;           use Strings_10;
5
6 procedure Main is
7     type Test_Case_Index is
8         (String_10_Long_Chk,
9          String_10_Short_Chk);
10
11    procedure Check (TC : Test_Case_Index) is
12        SL   : constant String := "And this is a long string just for testing...";
13        SS   : constant String := "Hey!";
14        S_10 : String_10;
15
16    begin
17        case TC is
18            when String_10_Long_Chk =>
19                S_10 := To_String_10 (SL);
20                Put_Line (String (S_10));
21            when String_10_Short_Chk =>
22                S_10 := (others => ' ');
23                S_10 := To_String_10 (SS);
24                Put_Line (String (S_10));
25        end case;
26    end Check;
27
28    begin
29        if Argument_Count < 1 then
30            Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting...");
31            return;
32        elsif Argument_Count > 1 then
33            Ada.Text_IO.Put_Line ("Ignoring additional arguments...");
34        end if;
35
```

(continues on next page)

(continued from previous page)

```

36   Check (Test_Case_Index'Value (Argument (1)));
37 end Main;
```

104.6.6 List of Names

Listing 73: names_ages.ads

```

1 package Names_Ages is
2
3   Max_People : constant Positive := 10;
4
5   subtype Name_Type is String (1 .. 50);
6
7   type Age_Type is new Natural;
8
9   type Person is record
10     Name : Name_Type;
11     Age  : Age_Type;
12   end record;
13
14  type People_Array is array (Positive range <>) of Person;
15
16  type People is record
17    People_A : People_Array (1 .. Max_People);
18    Last_Valid : Natural;
19  end record;
20
21  procedure Reset (P : in out People);
22
23  procedure Add (P    : in out People;
24                  Name : String);
25
26  function Get (P    : People;
27                 Name : String) return Age_Type;
28
29  procedure Update (P   : in out People;
30                     Name : String;
31                     Age  : Age_Type);
32
33  procedure Display (P : People);
34
35 end Names_Ages;
```

Listing 74: names_ages.adb

```

1 with Ada.Text_Io;      use Ada.Text_Io;
2 with Ada.Strings;      use Ada.Strings;
3 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
4
5 package body Names_Ages is
6
7   function To_Name_Type (S : String) return Name_Type is
8     S_Out : Name_Type := (others => ' ');
9   begin
10     for I in 1 .. Integer'Min (S'Last, Name_Type'Last) loop
11       S_Out (I) := S (I);
12     end loop;
13   end;
```

(continues on next page)

(continued from previous page)

```

14      return S_Out;
15  end To_Name_Type;
16
17  procedure Init (P    : in out Person;
18                  Name :          String) is
19  begin
20      P.Name := To_Name_Type (Name);
21      P.Age := 0;
22  end Init;
23
24  function Match (P    : Person;
25                  Name : String) return Boolean is
26  begin
27      return P.Name = To_Name_Type (Name);
28  end Match;
29
30  function Get (P : Person) return Age_Type is
31  begin
32      return P.Age;
33  end Get;
34
35  procedure Update (P   : in out Person;
36                     Age :          Age_Type) is
37  begin
38      P.Age := Age;
39  end Update;
40
41  procedure Display (P : Person) is
42  begin
43      Put_Line ("NAME: " & Trim (P.Name, Right));
44      Put_Line ("AGE: " & Age_Type'Image (P.Age));
45  end Display;
46
47  procedure Reset (P : in out People) is
48  begin
49      P.Last_Valid := 0;
50  end Reset;
51
52  procedure Add (P    : in out People;
53                  Name :          String) is
54  begin
55      P.Last_Valid := P.Last_Valid + 1;
56      Init (P.People_A (P.Last_Valid), Name);
57  end Add;
58
59  function Get (P    : People;
60                  Name : String) return Age_Type is
61  begin
62      for I in P.People_A'First .. P.Last_Valid loop
63          if Match (P.People_A (I), Name) then
64              return Get (P.People_A (I));
65          end if;
66      end loop;
67
68      return 0;
69  end Get;
70
71  procedure Update (P   : in out People;
72                     Name :          String;
73                     Age :          Age_Type) is
74  begin

```

(continues on next page)

(continued from previous page)

```

75   for I in P.Peoople_A'First .. P.Last_Valid loop
76     if Match (P.Peoople_A (I), Name) then
77       Update (P.Peoople_A (I), Age);
78     end if;
79   end loop;
80 end Update;

81
82 procedure Display (P : People) is
83 begin
84   Put_Line ("LIST OF NAMES:");
85   for I in P.Peoople_A'First .. P.Last_Valid loop
86     Display (P.Peoople_A (I));
87   end loop;
88 end Display;

89
90 end Names_Ages;

```

Listing 75: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Names_Ages;           use Names_Ages;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Names_Ages_Chk,
9       Get_Age_Chk);
10
11  procedure Check (TC : Test_Case_Index) is
12    P : People;
13  begin
14    case TC is
15      when Names_Ages_Chk =>
16        Reset (P);
17        Add (P, "John");
18        Add (P, "Patricia");
19        Add (P, "Josh");
20        Display (P);
21        Update (P, "John",     18);
22        Update (P, "Patricia", 35);
23        Update (P, "Josh",     53);
24        Display (P);
25      when Get_Age_Chk =>
26        Reset (P);
27        Add (P, "Peter");
28        Update (P, "Peter", 45);
29        Put_Line ("Peter is "
30                  & Age_Type'Image (Get (P, "Peter"))
31                  & " years old.");
32    end case;
33  end Check;
34
35  begin
36    if Argument_Count < 1 then
37      Ada.Text_IO.Put_Line ("ERROR: missing arguments! Exiting... ");
38      return;
39    elsif Argument_Count > 1 then
40      Ada.Text_IO.Put_Line ("Ignoring additional arguments... ");
41    end if;
42

```

(continues on next page)

(continued from previous page)

```

43   Check (Test_Case_Index'Value (Argument (1)));
44 end Main;

```

104.7 More About Types

104.7.1 Aggregate Initialization

Listing 76: aggregates.ads

```

1 package Aggregates is
2
3   type Rec is record
4     W : Integer := 10;
5     X : Integer := 11;
6     Y : Integer := 12;
7     Z : Integer := 13;
8   end record;
9
10  type Int_Arr is array (1 .. 20) of Integer;
11
12  procedure Init (R : out Rec);
13
14  procedure Init_Some (A : out Int_Arr);
15
16  procedure Init (A : out Int_Arr);
17
18 end Aggregates;

```

Listing 77: aggregates.adb

```

1 package body Aggregates is
2
3   procedure Init (R : out Rec) is
4   begin
5     R := (X      => 100,
6           Y      => 200,
7           others => <>);
8   end Init;
9
10  procedure Init_Some (A : out Int_Arr) is
11  begin
12    A := (1 .. 5 => 99,
13          others => 100);
14  end Init_Some;
15
16  procedure Init (A : out Int_Arr) is
17  begin
18    A := (others => 5);
19  end Init;
20
21 end Aggregates;

```

Listing 78: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;

```

(continues on next page)

(continued from previous page)

```

3   with Aggregates;           use Aggregates;
4
5
6 procedure Main is
7   -- Remark: the following line is not relevant.
8   F : array (1 .. 10) of Float := (others => 42.42)
9   with Unreferenced;
10
11 type Test_Case_Index is
12   (Default_Rec_Chk,
13    Init_Rec_Chk,
14    Init_Some_Arr_Chk,
15    Init_Arr_Chk);
16
17 procedure Check (TC : Test_Case_Index) is
18   A : Int_Arr;
19   R : Rec;
20   DR : constant Rec := (others => <>);
21 begin
22   case TC is
23     when Default_Rec_Chk =>
24       R := DR;
25       Put_Line ("Record Default:");
26       Put_Line ("W => " & Integer'Image (R.W));
27       Put_Line ("X => " & Integer'Image (R.X));
28       Put_Line ("Y => " & Integer'Image (R.Y));
29       Put_Line ("Z => " & Integer'Image (R.Z));
30     when Init_Rec_Chk =>
31       Init (R);
32       Put_Line ("Record Init:");
33       Put_Line ("W => " & Integer'Image (R.W));
34       Put_Line ("X => " & Integer'Image (R.X));
35       Put_Line ("Y => " & Integer'Image (R.Y));
36       Put_Line ("Z => " & Integer'Image (R.Z));
37     when Init_Some_Arr_Chk =>
38       Init_Some (A);
39       Put_Line ("Array Init_Some:");
40       for I in A'Range loop
41         Put_Line (Integer'Image (I) & " "
42                     & Integer'Image (A (I)));
43       end loop;
44     when Init_Arr_Chk =>
45       Init (A);
46       Put_Line ("Array Init:");
47       for I in A'Range loop
48         Put_Line (Integer'Image (I) & " "
49                     & Integer'Image (A (I)));
50       end loop;
51   end case;
52 end Check;
53
54 begin
55   if Argument_Count < 1 then
56     Put_Line ("ERROR: missing arguments! Exiting... ");
57     return;
58   elsif Argument_Count > 1 then
59     Put_Line ("Ignoring additional arguments... ");
60   end if;
61
62   Check (Test_Case_Index'Value (Argument (1)));
63 end Main;

```

104.7.2 Versioning

Listing 79: versioning.ads

```

1 package Versioning is
2
3     type Version is record
4         Major      : Natural;
5         Minor      : Natural;
6         Maintenance : Natural;
7     end record;
8
9     function Convert (V : Version) return String;
10
11    function Convert (V : Version) return Float;
12
13 end Versioning;

```

Listing 80: versioning.adb

```

1 with Ada.Strings; use Ada.Strings;
2 with Ada.Strings.Fixed; use Ada.Strings.Fixed;
3
4 package body Versioning is
5
6     function Image_Trim (N : Natural) return String is
7         S_N : constant String := Trim (Natural'Image (N), Left);
8     begin
9         return S_N;
10    end Image_Trim;
11
12    function Convert (V : Version) return String is
13        S_Major : constant String := Image_Trim (V.Major);
14        S_Minor : constant String := Image_Trim (V.Minor);
15        S_Maint : constant String := Image_Trim (V.Maintenance);
16    begin
17        return (S_Major & "." & S_Minor & "." & S_Maint);
18    end Convert;
19
20    function Convert (V : Version) return Float is
21    begin
22        return Float (V.Major) + (Float (V.Minor) / 10.0);
23    end Convert;
24
25 end Versioning;

```

Listing 81: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3
4 with Versioning;       use Versioning;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Ver_String_Chk,
9          Ver_Float_Chk);
10
11    procedure Check (TC : Test_Case_Index) is
12        V : constant Version := (1, 3, 23);
13    begin

```

(continues on next page)

(continued from previous page)

```

14  case TC is
15    when Ver_String_Chk =>
16      Put_Line (Convert (V));
17    when Ver_Float_Chk =>
18      Put_Line (Float'Image (Convert (V)));
19  end case;
20 end Check;

21
22 begin
23  if Argument_Count < 1 then
24    Put_Line ("ERROR: missing arguments! Exiting...");  

25    return;
26  elsif Argument_Count > 1 then
27    Put_Line ("Ignoring additional arguments...");  

28  end if;
29
30  Check (Test_Case_Index'Value (Argument (1)));
31 end Main;

```

104.7.3 Simple todo list

Listing 82: todo_lists.ads

```

1 package Todo_Lists is
2
3   type Todo_Item is access String;
4
5   type Todo_Items is array (Positive range <>) of Todo_Item;
6
7   type Todo_List (Max_Len : Natural) is record
8     Items : Todo_Items (1 .. Max_Len);
9     Last : Natural := 0;
10    end record;
11
12  procedure Add (Todos : in out Todo_List;
13                  Item  : String);
14
15  procedure Display (Todos : Todo_List);
16
17 end Todo_Lists;

```

Listing 83: todo_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Todo_Lists is
4
5   procedure Add (Todos : in out Todo_List;
6                  Item  : String) is
7   begin
8     if Todos.Last < Todos.Items'Last then
9       Todos.Last := Todos.Last + 1;
10      Todos.Items (Todos.Last) := new String'(Item);
11    else
12      Put_Line ("ERROR: list is full!");
13    end if;
14  end Add;
15

```

(continues on next page)

(continued from previous page)

```

16  procedure Display (Todos : Todo_List) is
17  begin
18      Put_Line ("TO-DO LIST");
19      for I in Todos.Items'First .. Todos.Last loop
20          Put_Line (Todos.Items (I).all);
21      end loop;
22  end Display;
23
24 end Todo_Lists;

```

Listing 84: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Todo_Lists;       use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List (10);
12     begin
13         case TC is
14             when Todo_List_Chk =>
15                 Add (T, "Buy milk");
16                 Add (T, "Buy tea");
17                 Add (T, "Buy present");
18                 Add (T, "Buy tickets");
19                 Add (T, "Pay electricity bill");
20                 Add (T, "Schedule dentist appointment");
21                 Add (T, "Call sister");
22                 Add (T, "Revise spreasheet");
23                 Add (T, "Edit entry page");
24                 Add (T, "Select new design");
25                 Add (T, "Create upgrade plan");
26                 Display (T);
27             end case;
28         end Check;
29
30     begin
31         if Argument_Count < 1 then
32             Put_Line ("ERROR: missing arguments! Exiting... ");
33             return;
34         elsif Argument_Count > 1 then
35             Put_Line ("Ignoring additional arguments... ");
36         end if;
37
38         Check (Test_Case_Index'Value (Argument (1)));
39     end Main;

```

104.7.4 Price list

Listing 85: price_lists.ads

```

1 package Price_Lists is
2
3     type Price_Type is delta 0.01 digits 12;
4
5     type Price_List_Array is array (Positive range <>) of Price_Type;
6
7     type Price_List (Max : Positive) is record
8         List : Price_List_Array (1 .. Max);
9         Last : Natural := 0;
10    end record;
11
12    type Price_Result (Ok : Boolean) is record
13        case Ok is
14            when False =>
15                null;
16            when True =>
17                Price : Price_Type;
18        end case;
19    end record;
20
21    procedure Reset (Prices : in out Price_List);
22
23    procedure Add (Prices : in out Price_List;
24                  Item   : Price_Type);
25
26    function Get (Prices : Price_List;
27                  IIdx   : Positive) return Price_Result;
28
29    procedure Display (Prices : Price_List);
30
31 end Price_Lists;

```

Listing 86: price_lists.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Price_Lists is
4
5     procedure Reset (Prices : in out Price_List) is
6     begin
7         Prices.Last := 0;
8     end Reset;
9
10    procedure Add (Prices : in out Price_List;
11                  Item   : Price_Type) is
12    begin
13        if Prices.Last < Prices.List'Last then
14            Prices.Last := Prices.Last + 1;
15            Prices.List (Prices.Last) := Item;
16        else
17            Put_Line ("ERROR: list is full!");
18        end if;
19    end Add;
20
21    function Get (Prices : Price_List;
22                  IIdx   : Positive) return Price_Result is
23    begin

```

(continues on next page)

(continued from previous page)

```

24    if (Idx >= Prices.List'First and then
25        Idx <= Prices.Last)      then
26        return Price_Result'(Ok     => True,
27                               Price => Prices.List (Idx));
28    else
29        return Price_Result'(Ok     => False);
30    end if;
31 end Get;

32
33 procedure Display (Prices : Price_List) is
34 begin
35    Put_Line ("PRICE LIST");
36    for I in Prices.List'First .. Prices.Last loop
37        Put_Line (Price_Type'Image (Prices.List (I)));
38    end loop;
39 end Display;
40
41 end Price_Lists;

```

Listing 87: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_Io;        use Ada.Text_Io;
3
4  with Price_Lists;       use Price_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Price_Type_Chk,
9           Price_List_Chk,
10          Price_List_Get_Chk);
11
12  procedure Check (TC : Test_Case_Index) is
13      L : Price_List (10);
14
15      procedure Local_Init_List is
16      begin
17          Reset (L);
18          Add (L, 1.45);
19          Add (L, 2.37);
20          Add (L, 3.21);
21          Add (L, 4.14);
22          Add (L, 5.22);
23          Add (L, 6.69);
24          Add (L, 7.77);
25          Add (L, 8.14);
26          Add (L, 9.99);
27          Add (L, 10.01);
28      end Local_Init_List;
29
30      procedure Get_Display (Idx : Positive) is
31          R : constant Price_Result := Get (L, Idx);
32      begin
33          Put_Line ("Attempt Get # " & Positive'Image (Idx));
34          if R.Ok then
35              Put_Line ("Element # " & Positive'Image (Idx)
36                      & " => "      & Price_Type'Image (R.Price));
37          else
38              declare
39                  begin
40                      Put_Line ("Element # " & Positive'Image (Idx)

```

(continues on next page)

(continued from previous page)

```

41           & " => "    & Price_Type'Image (R.Price));
42   exception
43     when others =>
44       Put_Line ("Element not available (as expected)");
45   end;
46 end if;

47
48 end Get_Display;

49

50 begin
51   case TC is
52     when Price_Type_Chk =>
53       Put_Line ("The delta value of Price_Type is "
54                 & Price_Type'Image (Price_Type'Delta) & ";");
55       Put_Line ("The minimum value of Price_Type is "
56                 & Price_Type'Image (Price_Type'First) & ";");
57       Put_Line ("The maximum value of Price_Type is "
58                 & Price_Type'Image (Price_Type'Last) & ";");
59     when Price_List_Chk =>
60       Local_Init_List;
61       Display (L);
62     when Price_List_Get_Chk =>
63       Local_Init_List;
64       Get_Display (5);
65       Get_Display (40);
66   end case;
67 end Check;

68
69 begin
70   if Argument_Count < 1 then
71     Put_Line ("ERROR: missing arguments! Exiting... ");
72     return;
73   elsif Argument_Count > 1 then
74     Put_Line ("Ignoring additional arguments... ");
75   end if;

76
77   Check (Test_Case_Index'Value (Argument (1)));
78 end Main;

```

104.8 Privacy

104.8.1 Directions

Listing 88: directions.ads

```

1 package Directions is
2
3   type Angle_Mod is mod 360;
4
5   type Direction is
6     (North,
7      Northwest,
8      West,
9      Southwest,
10     South,
11     Southeast,
12     East);

```

(continues on next page)

(continued from previous page)

```

13  function To_Direction (N : Angle_Mod) return Direction;
14
15  type Ext_Angle is private;
16
17  function To_Ext_Angle (N : Angle_Mod) return Ext_Angle;
18
19  procedure Display (N : Ext_Angle);
20
21
22  private
23
24  type Ext_Angle is record
25      Angle_Elem    : Angle_Mod;
26      Direction_Elem : Direction;
27  end record;
28
29 end Directions;
```

Listing 89: directions.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Directions is
4
5      procedure Display (N : Ext_Angle) is
6      begin
7          Put_Line ("Angle: "
8                  & Angle_Mod'Image (N.Angle_Elem)
9                  & " => "
10                 & Direction'Image (N.Direction_Elem)
11                 & ".");
12      end Display;
13
14      function To_Direction (N : Angle_Mod) return Direction is
15      begin
16          case N is
17              when 0          => return East;
18              when 1 .. 89    => return Northwest;
19              when 90         => return North;
20              when 91 .. 179  => return Northwest;
21              when 180        => return West;
22              when 181 .. 269 => return Southwest;
23              when 270        => return South;
24              when 271 .. 359 => return Southeast;
25          end case;
26      end To_Direction;
27
28      function To_Ext_Angle (N : Angle_Mod) return Ext_Angle is
29      begin
30          return (Angle_Elem    => N,
31                  Direction_Elem => To_Direction (N));
32      end To_Ext_Angle;
33
34 end Directions;
```

Listing 90: test_directions.adb

```

1  with Directions; use Directions;
2
3  procedure Test_Directions is
```

(continues on next page)

(continued from previous page)

```

4   type Ext_Angle_Array is array (Positive range <>) of Ext_Angle;
5
6   All_Directions : constant Ext_Angle_Array (1 .. 6)
7     := (To_Ext_Angle (0),
8           To_Ext_Angle (45),
9           To_Ext_Angle (90),
10          To_Ext_Angle (91),
11          To_Ext_Angle (180),
12          To_Ext_Angle (270));
13
14 begin
15   for I in All_Directions'Range loop
16     Display (All_Directions (I));
17   end loop;
18
19 end Test_Directions;

```

Listing 91: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Test_Directions;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Direction_Chk);
9
10   procedure Check (TC : Test_Case_Index) is
11   begin
12     case TC is
13       when Direction_Chk =>
14         Test_Directions;
15     end case;
16   end Check;
17
18 begin
19   if Argument_Count < 1 then
20     Put_Line ("ERROR: missing arguments! Exiting..."); return;
21   elsif Argument_Count > 1 then
22     Put_Line ("Ignoring additional arguments..."); end if;
23
24   Check (Test_Case_Index'Value (Argument (1)));
25
26 end Main;
27

```

104.8.2 Limited Strings

Listing 92: limited_strings.ads

```

1  package Limited_Strings is
2
3    type Lim_String is limited private;
4
5    function Init (S : String) return Lim_String;
6
7    function Init (Max : Positive) return Lim_String;

```

(continues on next page)

(continued from previous page)

```

8   procedure Put_Line (LS : Lim_String);
9
10  procedure Copy (From : Lim_String;
11                  To   : in out Lim_String);
12
13  function "=" (Ref, Dut : Lim_String) return Boolean;
14
15  private
16
17    type Lim_String is access String;
18
19  end Limited_Strings;
20

```

Listing 93: limited_strings.adb

```

1  with Ada.Text_IO;
2
3  package body Limited_Strings
4  is
5
6    function Init (S : String) return Lim_String is
7      LS : constant Lim_String := new String'(S);
8    begin
9      return LS;
10   end Init;
11
12  function Init (Max : Positive) return Lim_String is
13    LS : constant Lim_String := new String'(1 .. Max);
14  begin
15    LS.all := (others => '_');
16    return LS;
17  end Init;
18
19  procedure Put_Line (LS : Lim_String) is
20  begin
21    Ada.Text_Io.Put_Line (LS.all);
22  end Put_Line;
23
24  function Get_Min_Last (A, B : Lim_String) return Positive is
25  begin
26    return Positive'Min (A'Last, B'Last);
27  end Get_Min_Last;
28
29  procedure Copy (From : Lim_String;
30                  To   : in out Lim_String) is
31    Min_Last : constant Positive := Get_Min_Last (From, To);
32  begin
33    To (To'First .. Min_Last) := From (To'First .. Min_Last);
34    To (Min_Last + 1 .. To'Last) := (others => '_');
35  end;
36
37  function "=" (Ref, Dut : Lim_String) return Boolean is
38    Min_Last : constant Positive := Get_Min_Last (Ref, Dut);
39  begin
40    for I in Dut'First .. Min_Last loop
41      if Dut (I) /= Ref (I) then
42        return False;
43      end if;
44    end loop;
45

```

(continues on next page)

(continued from previous page)

```

46     return True;
47   end;
48
49 end Limited_Strings;

```

Listing 94: check_lim_string.adb

```

1  with Ada.Text_Io;      use Ada.Text_Io;
2
3  with Limited_Strings; use Limited_Strings;
4
5  procedure Check_Lim_String is
6    S : constant String := "-----";
7    S1 : constant Lim_String := Init ("Hello World");
8    S2 : constant Lim_String := Init (30);
9    S3 : Lim_String := Init (5);
10   S4 : Lim_String := Init (S & S & S);
11 begin
12   Put ("S1 => ");
13   Put_Line (S1);
14   Put ("S2 => ");
15   Put_Line (S2);
16
17   if S1 = S2 then
18     Put_Line ("S1 is equal to S2.");
19   else
20     Put_Line ("S1 isn't equal to S2.");
21   end if;
22
23   Copy (From => S1, To => S3);
24   Put ("S3 => ");
25   Put_Line (S3);
26
27   if S1 = S3 then
28     Put_Line ("S1 is equal to S3.");
29   else
30     Put_Line ("S1 isn't equal to S3.");
31   end if;
32
33   Copy (From => S1, To => S4);
34   Put ("S4 => ");
35   Put_Line (S4);
36
37   if S1 = S4 then
38     Put_Line ("S1 is equal to S4.");
39   else
40     Put_Line ("S1 isn't equal to S4.");
41   end if;
42 end Check_Lim_String;

```

Listing 95: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_Io;      use Ada.Text_Io;
3
4  with Check_Lim_String;
5
6  procedure Main is
7    type Test_Case_Index is
8      (Lim_String_Chk);

```

(continues on next page)

(continued from previous page)

```

9  procedure Check (TC : Test_Case_Index) is
10 begin
11   case TC is
12   when Lim_String_Chk =>
13     Check_Lim_String;
14   end case;
15 end Check;
16
17 begin
18   if Argument_Count < 1 then
19     Put_Line ("ERROR: missing arguments! Exiting...");  

20     return;
21   elsif Argument_Count > 1 then
22     Put_Line ("Ignoring additional arguments...");  

23   end if;
24
25   Check (Test_Case_Index'Value (Argument (1)));
26 end Main;
27

```

104.9 Generics

104.9.1 Display Array

Listing 96: display_array.ads

```

1 generic
2   type T_Range is range <>;
3   type T_Element is private;
4   type T_Array is array (T_Range range <>) of T_Element;
5   with function Image (E : T_Element) return String;
6   procedure Display_Array (Header : String;
7                           A      : T_Array);

```

Listing 97: display_array.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Display_Array (Header : String;
4                           A      : T_Array) is
5 begin
6   Put_Line (Header);
7   for I in A'Range loop
8     Put_Line (T_Range'Image (I) & ":" & Image (A (I)));
9   end loop;
10 end Display_Array;

```

Listing 98: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Array;
5
6 procedure Main is
7   type Test_Case_Index is (Int_Array_Chk,

```

(continues on next page)

(continued from previous page)

```

8                               Point_Array_Chk);

9
10  procedure Test_Int_Array is
11      type Int_Array is array (Positive range <>) of Integer;
12
13      procedure Display_Int_Array is new
14          Display_Array (T_Range => Positive,
15                          T_Element => Integer,
16                          T_Array    => Int_Array,
17                          Image      => Integer'Image);
18
19      A : constant Int_Array (1 .. 5) := (1, 2, 5, 7, 10);
20  begin
21      Display_Int_Array ("Integers", A);
22  end Test_Int_Array;
23
24  procedure Test_Point_Array is
25      type Point is record
26          X : Float;
27          Y : Float;
28      end record;
29
30      type Point_Array is array (Natural range <>) of Point;
31
32      function Image (P : Point) return String is
33  begin
34          return "(" & Float'Image (P.X)
35              & ", " & Float'Image (P.Y) & ")";
36  end Image;
37
38      procedure Display_Point_Array is new
39          Display_Array (T_Range    => Natural,
40                          T_Element  => Point,
41                          T_Array    => Point_Array,
42                          Image      => Image);
43
44      A : constant Point_Array (0 .. 3) := ((1.0, 0.5), (2.0, -0.5),
45                                              (5.0, 2.0), (-0.5, 2.0));
46  begin
47      Display_Point_Array ("Points", A);
48  end Test_Point_Array;
49
50  procedure Check (TC : Test_Case_Index) is
51  begin
52      case TC is
53          when Int_Array_Chk =>
54              Test_Int_Array;
55          when Point_Array_Chk =>
56              Test_Point_Array;
57      end case;
58  end Check;
59
60  begin
61      if Argument_Count < 1 then
62          Put_Line ("ERROR: missing arguments! Exiting... ");
63          return;
64      elsif Argument_Count > 1 then
65          Put_Line ("Ignoring additional arguments... ");
66      end if;
67
68      Check (Test_Case_Index'Value (Argument (1)));

```

(continues on next page)

(continued from previous page)

69 **end Main;**

104.9.2 Average of Array of Float

Listing 99: average.ads

```

1  generic
2    type T_Range is range <>;
3    type T_Element is digits <>;
4    type T_Array is array (T_Range range <>) of T_Element;
5  function Average (A : T_Array) return T_Element;

```

Listing 100: average.adb

```

1  function Average (A : T_Array) return T_Element is
2    Acc : Float := 0.0;
3  begin
4    for I in A'Range loop
5      Acc := Acc + Float (A (I));
6    end loop;
7
8    return T_Element (Acc / Float (A'Length));
9  end Average;

```

Listing 101: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_Io;       use Ada.Text_Io;
3
4  with Average;
5
6  procedure Main is
7    type Test_Case_Index is (Float_Array_Chk,
8                             Digits_7_Float_Array_Chk);
9
10 procedure Test_Float_Array is
11   type Float_Array is array (Positive range <>) of Float;
12
13   function Average_Float is new
14     Average (T_Range  => Positive,
15               T_Element => Float,
16               T_Array   => Float_Array);
17
18   A : constant Float_Array (1 .. 5) := (1.0, 3.0, 5.0, 7.5, -12.5);
19 begin
20   Put_Line ("Average: " & Float'Image (Average_Float (A)));
21 end Test_Float_Array;
22
23 procedure Test_Digits_7_Float_Array is
24   type Custom_Float is digits 7 range 0.0 .. 1.0;
25
26   type Float_Array is
27     array (Integer range <>) of Custom_Float;
28
29   function Average_Float is new
30     Average (T_Range  => Integer,
31               T_Element => Custom_Float,
32               T_Array   => Float_Array);

```

(continues on next page)

(continued from previous page)

```

33      A : constant Float_Array (-1 .. 3) := (0.5, 0.0, 1.0, 0.6, 0.5);
34
35  begin
36    Put_Line ("Average: "
37              & Custom_Float'Image (Average_Float (A)));
38  end Test_Digits_7_Float_Array;
39
40  procedure Check (TC : Test_Case_Index) is
41  begin
42    case TC is
43      when Float_Array_Chk =>
44        Test_Float_Array;
45      when Digits_7_Float_Array_Chk =>
46        Test_Digits_7_Float_Array;
47    end case;
48  end Check;
49
50 begin
51  if Argument_Count < 1 then
52    Put_Line ("ERROR: missing arguments! Exiting... ");
53    return;
54  elsif Argument_Count > 1 then
55    Put_Line ("Ignoring additional arguments... ");
56  end if;
57
58  Check (Test_Case_Index'Value (Argument (1)));
59 end Main;

```

104.9.3 Average of Array of Any Type

Listing 102: average.ads

```

1 generic
2   type T_Range is range <>;
3   type T_Element is private;
4   type T_Array is array (T_Range range <>) of T_Element;
5   with function To_Float (E : T_Element) return Float is <>;
6   function Average (A : T_Array) return Float;

```

Listing 103: average.adb

```

1 function Average (A : T_Array) return Float is
2   Acc : Float := 0.0;
3 begin
4   for I in A'Range loop
5     Acc := Acc + To_Float (A (I));
6   end loop;
7
8   return Acc / Float (A'Length);
9 end Average;

```

Listing 104: test_item.ads

```

1 procedure Test_Item;

```

Listing 105: test_item.adb

```

1  with Ada.Text_Io;      use Ada.Text_Io;
2
3  with Average;
4
5  procedure Test_Item is
6    package F_Io is new Ada.Text_Io.Float_Io (Float);
7
8    type Amount is delta 0.01 digits 12;
9
10   type Item is record
11     Quantity : Natural;
12     Price    : Amount;
13   end record;
14
15   type Item_Array is
16     array (Positive range <>) of Item;
17
18   function Get_Total (I : Item) return Float is
19     (Float (I.Quantity) * Float (I.Price));
20
21   function Get_Price (I : Item) return Float is
22     (Float (I.Price));
23
24   function Average_Total is new
25     Average (T_Range  => Positive,
26               T_Element => Item,
27               T_Array   => Item_Array,
28               To_Float  => Get_Total);
29
30   function Average_Price is new
31     Average (T_Range  => Positive,
32               T_Element => Item,
33               T_Array   => Item_Array,
34               To_Float  => Get_Price);
35
36   A : constant Item_Array (1 .. 4)
37   := ((Quantity => 5,    Price => 10.00),
38        (Quantity => 80,   Price => 2.50),
39        (Quantity => 40,   Price => 5.00),
40        (Quantity => 20,   Price => 12.50));
41
42 begin
43   Put ("Average per item & quantity: ");
44   F_Io.Put (Average_Total (A), 3, 2, 0);
45   New_Line;
46
47   Put ("Average price:           ");
48   F_Io.Put (Average_Price (A), 3, 2, 0);
49   New_Line;
50 end Test_Item;

```

Listing 106: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_Io;      use Ada.Text_Io;
3
4  with Test_Item;
5
6  procedure Main is

```

(continues on next page)

(continued from previous page)

```

7  type Test_Case_Index is (Item_Array_Chk);
8
9  procedure Check (TC : Test_Case_Index) is
10 begin
11    case TC is
12      when Item_Array_Chk =>
13        Test_Item;
14    end case;
15  end Check;
16
17 begin
18  if Argument_Count < 1 then
19    Put_Line ("ERROR: missing arguments! Exiting...");  

20    return;
21  elsif Argument_Count > 1 then
22    Put_Line ("Ignoring additional arguments...");  

23  end if;
24
25  Check (Test_Case_Index'Value (Argument (1)));
26 end Main;

```

104.9.4 Generic list

Listing 107: gen_list.ads

```

1 generic
2   type Item is private;
3   type Items is array (Positive range <>) of Item;
4   Name      : String;
5   List_Array : in out Items;
6   Last      : in out Natural;
7   with procedure Put (I : Item) is <>;
8 package Gen_List is
9
10  procedure Init;
11
12  procedure Add (I      :      Item;
13                  Status : out Boolean);
14
15  procedure Display;
16
17 end Gen_List;

```

Listing 108: gen_list.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Gen_List is
4
5   procedure Init is
6   begin
7     Last := List_Array'First - 1;
8   end Init;
9
10  procedure Add (I      :      Item;
11                  Status : out Boolean) is
12  begin
13    Status := Last < List_Array'Last;

```

(continues on next page)

(continued from previous page)

```

14      if Status then
15          Last := Last + 1;
16          List_Array (Last) := I;
17      end if;
18  end Add;
19
20  procedure Display is
21  begin
22      Put_Line (Name);
23      for I in List_Array'First .. Last loop
24          Put (List_Array (I));
25          New_Line;
26      end loop;
27  end Display;
28
29 end Gen_List;
30

```

Listing 109: test_int.ads

```

1  procedure Test_Int;

```

Listing 110: test_int.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  with Gen_List;
4
5  procedure Test_Int is
6
7      procedure Put (I : Integer) is
8      begin
9          Ada.Text_IO.Put (Integer'Image (I));
10     end Put;
11
12     type Integer_Array is array (Positive range <>) of Integer;
13
14     A : Integer_Array (1 .. 3);
15     L : Natural;
16
17     package Int_List is new
18         Gen_List (Item        => Integer,
19                    Items       => Integer_Array,
20                    Name        => "List of integers",
21                    List_Array  => A,
22                    Last        => L);
23
24     Success : Boolean;
25
26     procedure Display_Add_Success (Success : Boolean) is
27     begin
28         if Success then
29             Put_Line ("Added item successfully!");
30         else
31             Put_Line ("Couldn't add item!");
32         end if;
33
34     end Display_Add_Success;
35
36 begin

```

(continues on next page)

(continued from previous page)

```

37 Int_List.Init;
38
39 Int_List.Add (2, Success);
40 Display_Add_Success (Success);
41
42 Int_List.Add (5, Success);
43 Display_Add_Success (Success);
44
45 Int_List.Add (7, Success);
46 Display_Add_Success (Success);
47
48 Int_List.Add (8, Success);
49 Display_Add_Success (Success);
50
51 Int_List.Display;
52 end Test_Int;

```

Listing 111: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Test_Int;
5
6 procedure Main is
7   type Test_Case_Index is (Int_Chk);
8
9   procedure Check (TC : Test_Case_Index) is
10 begin
11   case TC is
12     when Int_Chk =>
13       Test_Int;
14   end case;
15 end Check;
16
17 begin
18   if Argument_Count < 1 then
19     Put_Line ("ERROR: missing arguments! Exiting...");
20     return;
21   elsif Argument_Count > 1 then
22     Put_Line ("Ignoring additional arguments...");
23   end if;
24
25   Check (Test_Case_Index'Value (Argument (1)));
26 end Main;

```

104.10 Exceptions

104.10.1 Uninitialized Value

Listing 112: options.ads

```

1 package Options is
2
3   type Option is (Uninitialized,
4                  Option_1,

```

(continues on next page)

(continued from previous page)

```

5           Option_2,
6           Option_3);

7
8   Uninitialized_Value : exception;
9
10  function Image (0 : Option) return String;
11
12 end Options;
```

Listing 113: options.adb

```

1 package body Options is
2
3   function Image (0 : Option) return String is
4     begin
5       case 0 is
6         when Uninitialized =>
7           raise Uninitialized_Value with "Uninitialized value detected!";
8         when others =>
9           return Option'Image (0);
10      end case;
11    end Image;
12
13 end Options;
```

Listing 114: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3 with Ada.Exceptions;  use Ada.Exceptions;
4
5 with Options;          use Options;
6
7 procedure Main is
8   type Test_Case_Index is
9     (Options_Chk);
10
11 procedure Check (TC : Test_Case_Index) is
12
13   procedure Check (0 : Option) is
14     begin
15       Put_Line (Image (0));
16     exception
17       when E : Uninitialized_Value =>
18         Put_Line (Exception_Message (E));
19     end Check;
20
21   begin
22     case TC is
23     when Options_Chk =>
24       for O in Option loop
25         Check (O);
26       end loop;
27     end case;
28   end Check;
29
30 begin
31   if Argument_Count < 1 then
32     Put_Line ("ERROR: missing arguments! Exiting...");
```

(continues on next page)

(continued from previous page)

```

34  elsif Argument_Count > 1 then
35      Put_Line ("Ignoring additional arguments..."); 
36  end if;
37
38  Check (Test_Case_Index'Value (Argument (1)));
39 end Main;

```

104.10.2 Numerical Exception

Listing 115: tests.ads

```

1 package Tests is
2
3     type Test_ID is (Test_1, Test_2);
4
5     Custom_Exception : exception;
6
7     procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;

```

Listing 116: tests.adb

```

1 package body Tests is
2
3     pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5     procedure Num_Exception_Test (ID : Test_ID) is
6         A, B, C : Integer;
7     begin
8         case ID is
9             when Test_1 =>
10                 A := Integer'Last;
11                 B := Integer'Last;
12                 C := A + B;
13             when Test_2 =>
14                 raise Custom_Exception with "Custom_Exception raised!";
15         end case;
16     end Num_Exception_Test;
17
18     pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;

```

Listing 117: check_exception.adb

```

1 with Tests;          use Tests;
2
3 with Ada.Text_IO;    use Ada.Text_IO;
4 with Ada.Exceptions; use Ada.Exceptions;
5
6 procedure Check_Exception (ID : Test_ID) is
7 begin
8     Num_Exception_Test (ID);
9 exception
10    when Constraint_Error =>
11        Put_Line ("Constraint_Error detected!");
12    when E : others =>

```

(continues on next page)

(continued from previous page)

```
13     Put_Line (Exception_Message (E));
14 end Check_Exception;
```

Listing 118: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3  with Ada.Exceptions;   use Ada.Exceptions;
4
5  with Tests;           use Tests;
6  with Check_Exception;
7
8  procedure Main is
9    type Test_Case_Index is
10       (Exception_1_Chk,
11        Exception_2_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14
15    procedure Check_Handle_Exception (ID : Test_ID) is
16    begin
17      Check_Exception (ID);
18    exception
19      when Constraint_Error =>
20        Put_Line ("Constraint_Error"
21                  & " (raised by Check_Exception) detected!");
22      when E : others =>
23        Put_Line (Exception_Name (E)
24                  & " (raised by Check_Exception) detected!");
25    end Check_Handle_Exception;
26
27  begin
28    case TC is
29    when Exception_1_Chk =>
30      Check_Handle_Exception (Test_1);
31    when Exception_2_Chk =>
32      Check_Handle_Exception (Test_2);
33    end case;
34  end Check;
35
36  begin
37    if Argument_Count < 1 then
38      Put_Line ("ERROR: missing arguments! Exiting... ");
39      return;
40    elsif Argument_Count > 1 then
41      Put_Line ("Ignoring additional arguments... ");
42    end if;
43
44    Check (Test_Case_Index'Value (Argument (1)));
45  end Main;
```

104.10.3 Re-raising Exceptions

Listing 119: tests.ads

```

1 package Tests is
2
3   type Test_ID is (Test_1, Test_2);
4
5   Custom_Exception, Another_Exception : exception;
6
7   procedure Num_Exception_Test (ID : Test_ID);
8
9 end Tests;

```

Listing 120: tests.adb

```

1 package body Tests is
2
3   pragma Warnings (Off, "variable ""C"" is assigned but never read");
4
5   procedure Num_Exception_Test (ID : Test_ID) is
6     A, B, C : Integer;
7   begin
8     case ID is
9       when Test_1 =>
10        A := Integer'Last;
11        B := Integer'Last;
12        C := A + B;
13       when Test_2 =>
14         raise Custom_Exception with "Custom_Exception raised!";
15     end case;
16   end Num_Exception_Test;
17
18   pragma Warnings (On, "variable ""C"" is assigned but never read");
19
20 end Tests;

```

Listing 121: check_exception.ads

```

1 with Tests; use Tests;
2
3 procedure Check_Exception (ID : Test_ID);

```

Listing 122: check_exception.adb

```

1 with Ada.Text_IO;      use Ada.Text_IO;
2 with Ada.Exceptions; use Ada.Exceptions;
3
4 procedure Check_Exception (ID : Test_ID) is
5 begin
6   Num_Exception_Test (ID);
7 exception
8   when Constraint_Error =>
9     Put_Line ("Constraint_Error detected!");
10    raise;
11   when E : others =>
12     Put_Line (Exception_Message (E));
13     raise Another_Exception;
14 end Check_Exception;

```

Listing 123: main.adb

```
1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;         use Ada.Text_IO;
3  with Ada.Exceptions;    use Ada.Exceptions;
4
5  with Tests;             use Tests;
6  with Check_Exception;
7
8  procedure Main is
9      type Test_Case_Index is
10         (Exception_1_Chk,
11          Exception_2_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14
15      procedure Check_Handle_Exception (ID : Test_ID) is
16          begin
17              Check_Exception (ID);
18          exception
19              when Constraint_Error =>
20                  Put_Line ("Constraint_Error"
21                            & " (raised by Check_Exception) detected!");
22              when E : others =>
23                  Put_Line (Exception_Name (E)
24                            & " (raised by Check_Exception) detected!");
25          end Check_Handle_Exception;
26
27      begin
28          case TC is
29              when Exception_1_Chk =>
30                  Check_Handle_Exception (Test_1);
31              when Exception_2_Chk =>
32                  Check_Handle_Exception (Test_2);
33          end case;
34      end Check;
35
36  begin
37      if Argument_Count < 1 then
38          Put_Line ("ERROR: missing arguments! Exiting... ");
39          return;
40      elsif Argument_Count > 1 then
41          Put_Line ("Ignoring additional arguments... ");
42      end if;
43
44      Check (Test_Case_Index'Value (Argument (1)));
45  end Main;
```

104.11 Tasking

104.11.1 Display Service

Listing 124: display_services.ads

```
1  package Display_Services is
2
3      task type Display_Service is
```

(continues on next page)

(continued from previous page)

```

4   entry Display (S : String);
5   entry Display (I : Integer);
6 end Display_Service;
7
8 end Display_Services;

```

Listing 125: display_services.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Display_Services is
4
5   task body Display_Service is
6     begin
7       loop
8         select
9           accept Display (S : String) do
10              Put_Line (S);
11            end Display;
12        or
13          accept Display (I : Integer) do
14              Put_Line (Integer'Image (I));
15            end Display;
16        or
17          terminate;
18        end select;
19      end loop;
20    end Display_Service;
21
22 end Display_Services;

```

Listing 126: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Display_Services; use Display_Services;
5
6 procedure Main is
7   type Test_Case_Index is (Display_Service_Chk);
8
9   procedure Check (TC : Test_Case_Index) is
10     Display : Display_Service;
11   begin
12     case TC is
13       when Display_Service_Chk =>
14         Display.Display ("Hello");
15         delay 0.5;
16         Display.Display ("Hello again");
17         delay 0.5;
18         Display.Display (55);
19         delay 0.5;
20     end case;
21   end Check;
22
23 begin
24   if Argument_Count < 1 then
25     Put_Line ("ERROR: missing arguments! Exiting... ");
26     return;
27   elsif Argument_Count > 1 then

```

(continues on next page)

(continued from previous page)

```

28     Put_Line ("Ignoring additional arguments...");  

29   end if;  

30  

31   Check (Test_Case_Index'Value (Argument (1)));  

32 end Main;

```

104.11.2 Event Manager

Listing 127: event_managers.ads

```

1  with Ada.Real_Time; use Ada.Real_Time;  

2  

3  package Event_Managers is  

4  

5    task type Event_Manager is  

6      entry Start (ID : Natural);  

7      entry Event (T : Time);  

8    end Event_Manager;  

9  

10 end Event_Managers;

```

Listing 128: event_managers.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;  

2  

3  package body Event_Managers is  

4  

5    task body Event_Manager is  

6      Event_ID      : Natural := 0;  

7      Event_Delay : Time;  

8    begin  

9      accept Start (ID : Natural) do  

10        Event_ID := ID;  

11      end Start;  

12  

13      accept Event (T : Time) do  

14        Event_Delay := T;  

15      end Event;  

16  

17      delay until Event_Delay;  

18  

19      Put_Line ("Event #" & Natural'Image (Event_ID));  

20    end Event_Manager;  

21  

22 end Event_Managers;

```

Listing 129: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;  

2  with Ada.Text_IO;      use Ada.Text_IO;  

3  

4  with Event_Managers;  use Event_Managers;  

5  with Ada.Real_Time;   use Ada.Real_Time;  

6  

7  procedure Main is  

8    type Test_Case_Index is (Event_Manager_Chk);  

9  

10   procedure Check (TC : Test_Case_Index) is

```

(continues on next page)

(continued from previous page)

```

11   Ev_Mng : array (1 .. 5) of Event_Manager;
12 begin
13   case TC is
14     when Event_Manager_Clk =>
15       for I in Ev_Mng'Range loop
16         Ev_Mng (I).Start (I);
17       end loop;
18     Ev_Mng (1).Event (Clock + Seconds (5));
19     Ev_Mng (2).Event (Clock + Seconds (3));
20     Ev_Mng (3).Event (Clock + Seconds (1));
21     Ev_Mng (4).Event (Clock + Seconds (2));
22     Ev_Mng (5).Event (Clock + Seconds (4));
23   end case;
24 end Check;

25
26 begin
27   if Argument_Count < 1 then
28     Put_Line ("ERROR: missing arguments! Exiting...");
29     return;
30   elsif Argument_Count > 1 then
31     Put_Line ("Ignoring additional arguments...");
32   end if;
33
34   Check (Test_Case_Index'Value (Argument (1)));
35 end Main;

```

104.11.3 Generic Protected Queue

Listing 130: gen_queues.ads

```

1 generic
2   type Queue_Index is mod <>;
3   type T is private;
4 package Gen_Queues is
5
6   type Queue_Array is array (Queue_Index) of T;
7
8   protected type Queue is
9     function Empty return Boolean;
10    function Full return Boolean;
11    entry Push (V : T);
12    entry Pop (V : out T);
13  private
14    N : Natural := 0;
15    Idx : Queue_Index := Queue_Array'First;
16    A : Queue_Array;
17  end Queue;
18
19 end Gen_Queues;

```

Listing 131: gen_queues.adb

```

1 package body Gen_Queues is
2
3   protected body Queue is
4
5     function Empty return Boolean is
6       (N = 0);

```

(continues on next page)

(continued from previous page)

```

7   function Full return Boolean is
8     (N = A'Length);
9
10  entry Push (V : T) when not Full is
11    begin
12      A (Idx) := V;
13
14      Idx := Idx + 1;
15      N   := N + 1;
16    end Push;
17
18  entry Pop (V : out T) when not Empty is
19    begin
20      N := N - 1;
21
22      V := A (Idx - Queue_Index (N) - 1);
23    end Pop;
24
25  end Queue;
26
27
28 end Gen_Queues;
```

Listing 132: queue_tests.ads

```

1 package Queue_Tests is
2
3   procedure Simple_Test;
4
5   procedure Concurrent_Test;
6
7 end Queue_Tests;
```

Listing 133: queue_tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 with Gen_Queues;
4
5 package body Queue_Tests is
6
7   Max : constant := 10;
8   type Queue_Mod is mod Max;
9
10  procedure Simple_Test is
11    package Queues_Float is new Gen_Queues (Queue_Mod, Float);
12
13    Q_F : Queues_Float.Queue;
14    V   : Float;
15
16  begin
17    V := 10.0;
18    while not Q_F.Full loop
19      Q_F.Push (V);
20      V := V + 1.5;
21    end loop;
22
23    while not Q_F.Empty loop
24      Q_F.Pop (V);
25      Put_Line ("Value from queue: " & Float'Image (V));
26    end loop;
```

(continues on next page)

(continued from previous page)

```

26 end Simple_Test;
27
28 procedure Concurrent_Test is
29   package Queues_Integer is new Gen_Queues (Queue_Mod, Integer);
30
31   Q_I : Queues_Integer.Queue;
32
33   task T_Producer;
34   task T_Consumer;
35
36   task body T_Producer is
37     V : Integer := 100;
38   begin
39     for I in 1 .. 2 * Max loop
40       Q_I.Push (V);
41       V := V + 1;
42     end loop;
43   end T_Producer;
44
45   task body T_Consumer is
46     V : Integer;
47   begin
48     delay 1.5;
49
50     while not Q_I.Empty loop
51       Q_I.Pop (V);
52       Put_Line ("Value from queue: " & Integer'Image (V));
53       delay 0.2;
54     end loop;
55   end T_Consumer;
56 begin
57   null;
58 end Concurrent_Test;
59
60 end Queue_Tests;

```

Listing 134: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Queue_Tests;      use Queue_Tests;
5
6 procedure Main is
7   type Test_Case_Index is (Simple_Queue_Chk,
8                             Concurrent_Queue_Chk);
9
10  procedure Check (TC : Test_Case_Index) is
11  begin
12    case TC is
13      when Simple_Queue_Chk =>
14        Simple_Test;
15      when Concurrent_Queue_Chk =>
16        Concurrent_Test;
17    end case;
18  end Check;
19
20 begin
21  if Argument_Count < 1 then
22    Put_Line ("ERROR: missing arguments! Exiting...");
```

(continues on next page)

(continued from previous page)

```

24    elsif Argument_Count > 1 then
25        Put_Line ("Ignoring additional arguments..."); 
26    end if;
27
28    Check (Test_Case_Index'Value (Argument (1)));
29 end Main;

```

104.12 Design by contracts

104.12.1 Price Range

Listing 135: prices.ads

```

1 package Prices is
2
3     type Amount is delta 10.0 ** (-2) digits 12;
4
5     -- subtype Price is Amount range 0.0 .. Amount'Last;
6
7     subtype Price is Amount
8         with Static_Predicate => Price >= 0.0;
9
10 end Prices;

```

Listing 136: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Prices;           use Prices;
6
7 procedure Main is
8
9     type Test_Case_Index is
10        (Price_Range_Chk);
11
12     procedure Check (TC : Test_Case_Index) is
13
14         procedure Check_Range (A : Amount) is
15             P : constant Price := A;
16         begin
17             Put_Line ("Price: " & Price'Image (P));
18         end Check_Range;
19
20     begin
21         case TC is
22             when Price_Range_Chk =>
23                 Check_Range (-2.0);
24         end case;
25     exception
26         when Constraint_Error =>
27             Put_Line ("Constraint_Error detected (NOT as expected).");
28         when Assert_Failure =>
29             Put_Line ("Assert_Failure detected (as expected).");
30     end Check;

```

(continues on next page)

(continued from previous page)

```

31 begin
32   if Argument_Count < 1 then
33     Put_Line ("ERROR: missing arguments! Exiting... ");
34     return;
35   elsif Argument_Count > 1 then
36     Put_Line ("Ignoring additional arguments... ");
37   end if;
38
39   Check (Test_Case_Index'Value (Argument (1)));
40
41 end Main;

```

104.12.2 Pythagorean Theorem: Predicate

Listing 137: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is record
6     H      : Length := 0;
7     -- Hypotenuse
8     C1, C2 : Length := 0;
9     -- Catheti / legs
10    end record
11   with Dynamic_Predicate => H * H = C1 * C1 + C2 * C2;
12
13   function Init (H, C1, C2 : Length) return Right_Triangle is
14     ((H, C1, C2));
15
16 end Triangles;

```

Listing 138: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 139: triangles-io.adb

```

1 package body Triangles.IO is
2
3   function Image (T : Right_Triangle) return String is
4     ("(" & Length'Image (T.H)
5      & ", " & Length'Image (T.C1)
6      & ", " & Length'Image (T.C2)
7      & ")");
8
9 end Triangles.IO;

```

Listing 140: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;

```

(continues on next page)

(continued from previous page)

```
3  with System.Assertions; use System.Assertions;
4
5  with Triangles;           use Triangles;
6  with Triangles.IO;        use Triangles.IO;
7
8  procedure Main is
9
10   type Test_Case_Index is
11     (Triangle_8_6_Pass_Chk,
12      Triangle_8_6_Fail_Chk,
13      Triangle_10_24_Pass_Chk,
14      Triangle_10_24_Fail_Chk,
15      Triangle_18_24_Pass_Chk,
16      Triangle_18_24_Fail_Chk);
17
18  procedure Check (TC : Test_Case_Index) is
19
20    procedure Check_Triangle (H, C1, C2 : Length) is
21      T : Right_Triangle;
22    begin
23      T := Init (H, C1, C2);
24      Put_Line (Image (T));
25    exception
26      when Constraint_Error =>
27        Put_Line ("Constraint_Error detected (NOT as expected).");
28      when Assert_Failure =>
29        Put_Line ("Assert_Failure detected (as expected).");
30    end Check_Triangle;
31
32  begin
33    case TC is
34      when Triangle_8_6_Pass_Chk  => Check_Triangle (10, 8, 6);
35      when Triangle_8_6_Fail_Chk  => Check_Triangle (12, 8, 6);
36      when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37      when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38      when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39      when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40    end case;
41  end Check;
42
43  begin
44    if Argument_Count < 1 then
45      Put_Line ("ERROR: missing arguments! Exiting... ");
46      return;
47    elsif Argument_Count > 1 then
48      Put_Line ("Ignoring additional arguments... ");
49    end if;
50
51    Check (Test_Case_Index'Value (Argument (1)));
52  end Main;
```

104.12.3 Pythagorean Theorem: Precondition

Listing 141: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is record
6     H      : Length := 0;
7     -- Hypotenuse
8     C1, C2 : Length := 0;
9     -- Catheti / legs
10    end record;
11
12   function Init (H, C1, C2 : Length) return Right_Triangle is
13     ((H, C1, C2))
14     with Pre => H * H = C1 * C1 + C2 * C2;
15
16 end Triangles;

```

Listing 142: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 143: triangles-io.adb

```

1 package body Triangles.IO is
2
3   function Image (T : Right_Triangle) return String is
4     ("(" & Length'Image (T.H)
5      & ", " & Length'Image (T.C1)
6      & ", " & Length'Image (T.C2)
7      & ")");
8
9 end Triangles.IO;

```

Listing 144: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10  type Test_Case_Index is
11    (Triangle_8_6_Pass_Chk,
12     Triangle_8_6_Fail_Chk,
13     Triangle_10_24_Pass_Chk,
14     Triangle_10_24_Fail_Chk,
15     Triangle_18_24_Pass_Chk,
16     Triangle_18_24_Fail_Chk);
17
18  procedure Check (TC : Test_Case_Index) is

```

(continues on next page)

(continued from previous page)

```

19
20      procedure Check_Triangle (H, C1, C2 : Length) is
21          T : Right_Triangle;
22      begin
23          T := Init (H, C1, C2);
24          Put_Line (Image (T));
25      exception
26          when Constraint_Error =>
27              Put_Line ("Constraint_Error detected (NOT as expected).");
28          when Assert_Failure =>
29              Put_Line ("Assert_Failure detected (as expected).");
30      end Check_Triangle;
31
32  begin
33      case TC is
34          when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35          when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36          when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37          when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38          when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39          when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40      end case;
41  end Check;
42
43  begin
44      if Argument_Count < 1 then
45          Put_Line ("ERROR: missing arguments! Exiting... ");
46          return;
47      elsif Argument_Count > 1 then
48          Put_Line ("Ignoring additional arguments... ");
49      end if;
50
51      Check (Test_Case_Index'Value (Argument (1)));
52  end Main;

```

104.12.4 Pythagorean Theorem: Postcondition

Listing 145: triangles.ads

```

1 package Triangles is
2
3     subtype Length is Integer;
4
5     type Right_Triangle is record
6         H      : Length := 0;
7         -- Hypotenuse
8         C1, C2 : Length := 0;
9         -- Catheti / legs
10    end record;
11
12    function Init (H, C1, C2 : Length) return Right_Triangle is
13        ((H, C1, C2))
14        with Post => (Init'Result.H * Init'Result.H
15                      = Init'Result.C1 * Init'Result.C1
16                      + Init'Result.C2 * Init'Result.C2);
17
18  end Triangles;

```

Listing 146: triangles-io.ads

```

1 package Triangles.IO is
2
3     function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 147: triangles-io.adb

```

1 package body Triangles.IO is
2
3     function Image (T : Right_Triangle) return String is
4         ("(" & Length'Image (T.H)
5          & ", " & Length'Image (T.C1)
6          & ", " & Length'Image (T.C2)
7          & ")");
8
9 end Triangles.IO;

```

Listing 148: main.adb

```

1 with Ada.Command_Line;  use Ada.Command_Line;
2 with Ada.Text_IO;       use Ada.Text_IO;
3 with System.Assertions; use System.Assertions;
4
5 with Triangles;         use Triangles;
6 with Triangles.IO;      use Triangles.IO;
7
8 procedure Main is
9
10    type Test_Case_Index is
11        (Triangle_8_6_Pass_Chk,
12         Triangle_8_6_Fail_Chk,
13         Triangle_10_24_Pass_Chk,
14         Triangle_10_24_Fail_Chk,
15         Triangle_18_24_Pass_Chk,
16         Triangle_18_24_Fail_Chk);
17
18    procedure Check (TC : Test_Case_Index) is
19
20        procedure Check_Triangle (H, C1, C2 : Length) is
21            T : Right_Triangle;
22        begin
23            T := Init (H, C1, C2);
24            Put_Line (Image (T));
25        exception
26            when Constraint_Error =>
27                Put_Line ("Constraint_Error detected (NOT as expected).");
28            when Assert_Failure =>
29                Put_Line ("Assert_Failure detected (as expected).");
30        end Check_Triangle;
31
32    begin
33        case TC is
34            when Triangle_8_6_Pass_Chk => Check_Triangle (10, 8, 6);
35            when Triangle_8_6_Fail_Chk => Check_Triangle (12, 8, 6);
36            when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37            when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38            when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39            when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);

```

(continues on next page)

(continued from previous page)

```

40      end case;
41  end Check;

42
43 begin
44  if Argument_Count < 1 then
45    Put_Line ("ERROR: missing arguments! Exiting... ");
46    return;
47  elsif Argument_Count > 1 then
48    Put_Line ("Ignoring additional arguments... ");
49  end if;
50
51  Check (Test_Case_Index'Value (Argument (1)));
52 end Main;

```

104.12.5 Pythagorean Theorem: Type Invariant

Listing 149: triangles.ads

```

1 package Triangles is
2
3   subtype Length is Integer;
4
5   type Right_Triangle is private
6     with Type_Invariant => Check (Right_Triangle);
7
8   function Check (T : Right_Triangle) return Boolean;
9
10  function Init (H, C1, C2 : Length) return Right_Triangle;
11
12 private
13
14  type Right_Triangle is record
15    H      : Length := 0;
16    -- Hypotenuse
17    C1, C2 : Length := 0;
18    -- Catheti / legs
19  end record;
20
21  function Init (H, C1, C2 : Length) return Right_Triangle is
22    ((H, C1, C2));
23
24  function Check (T : Right_Triangle) return Boolean is
25    (T.H * T.H = T.C1 * T.C1 + T.C2 * T.C2);
26
27 end Triangles;

```

Listing 150: triangles-io.ads

```

1 package Triangles.IO is
2
3   function Image (T : Right_Triangle) return String;
4
5 end Triangles.IO;

```

Listing 151: triangles-io.adb

```

1 package body Triangles.IO is
2

```

(continues on next page)

(continued from previous page)

```

3   function Image (T : Right_Triangle) return String is
4     ("( " & Length'Image (T.H)
5       & ", " & Length'Image (T.C1)
6       & ", " & Length'Image (T.C2)
7       & ")");
8
9 end Triangles.IO;

```

Listing 152: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3  with System.Assertions; use System.Assertions;
4
5  with Triangles;         use Triangles;
6  with Triangles.IO;      use Triangles.IO;
7
8  procedure Main is
9
10 type Test_Case_Index is
11   (Triangle_8_6_Pass_Chk,
12    Triangle_8_6_Fail_Chk,
13    Triangle_10_24_Pass_Chk,
14    Triangle_10_24_Fail_Chk,
15    Triangle_18_24_Pass_Chk,
16    Triangle_18_24_Fail_Chk);
17
18 procedure Check (TC : Test_Case_Index) is
19
20   procedure Check_Triangle (H, C1, C2 : Length) is
21     T : Right_Triangle;
22   begin
23     T := Init (H, C1, C2);
24     Put_Line (Image (T));
25   exception
26     when Constraint_Error =>
27       Put_Line ("Constraint_Error detected (NOT as expected).");
28     when Assert_Failure =>
29       Put_Line ("Assert_Failure detected (as expected).");
30   end Check_Triangle;
31
32 begin
33   case TC is
34     when Triangle_8_6_Pass_Chk  => Check_Triangle (10, 8, 6);
35     when Triangle_8_6_Fail_Chk  => Check_Triangle (12, 8, 6);
36     when Triangle_10_24_Pass_Chk => Check_Triangle (26, 10, 24);
37     when Triangle_10_24_Fail_Chk => Check_Triangle (12, 10, 24);
38     when Triangle_18_24_Pass_Chk => Check_Triangle (30, 18, 24);
39     when Triangle_18_24_Fail_Chk => Check_Triangle (32, 18, 24);
40   end case;
41 end Check;
42
43 begin
44   if Argument_Count < 1 then
45     Put_Line ("ERROR: missing arguments! Exiting...");
46     return;
47   elsif Argument_Count > 1 then
48     Put_Line ("Ignoring additional arguments...");
49   end if;
50
51   Check (Test_Case_Index'Value (Argument (1)));

```

(continues on next page)

52 **end Main;**

104.12.6 Primary Colors

Listing 153: color_types.ads

```

1  package Color_Types is
2
3    type HTML_Color is
4      (Salmon,
5       Firebrick,
6       Red,
7       Darkred,
8       Lime,
9       Forestgreen,
10      Green,
11      Darkgreen,
12      Blue,
13      Mediumblue,
14      Darkblue);
15
16    subtype Int_Color is Integer range 0 .. 255;
17
18    function Image (I : Int_Color) return String;
19
20    type RGB is record
21      Red   : Int_Color;
22      Green : Int_Color;
23      Blue  : Int_Color;
24    end record;
25
26    function To_RGB (C : HTML_Color) return RGB;
27
28    function Image (C : RGB) return String;
29
30    type HTML_Color_RGB_Array is array (HTML_Color) of RGB;
31
32    To_RGB_Lookup_Table : constant HTML_Color_RGB_Array
33      := (Salmon      => (16#FA#, 16#80#, 16#72#),
34           Firebrick   => (16#B2#, 16#22#, 16#22#),
35           Red         => (16#FF#, 16#00#, 16#00#),
36           Darkred     => (16#8B#, 16#00#, 16#00#),
37           Lime        => (16#00#, 16#FF#, 16#00#),
38           Forestgreen => (16#22#, 16#8B#, 16#22#),
39           Green        => (16#00#, 16#80#, 16#00#),
40           Darkgreen   => (16#00#, 16#64#, 16#00#),
41           Blue         => (16#00#, 16#00#, 16#FF#),
42           Mediumblue  => (16#00#, 16#00#, 16#CD#),
43           Darkblue    => (16#00#, 16#00#, 16#8B#));
44
45    subtype HTML_RGB_Color is HTML_Color
46      with Static_Predicate => HTML_RGB_Color in Red | Green | Blue;
47
48    function To_Int_Color (C : HTML_Color;
49                          S : HTML_RGB_Color) return Int_Color;
50    -- Convert to hexadecimal value for the selected RGB component S
51
52  end Color_Types;

```

Listing 154: color_types.adb

```

1  with Ada.Integer_Text_IO;
2
3  package body Color_Types is
4
5      function To_RGB (C : HTML_Color) return RGB is
6  begin
7      return To_RGB_Lookup_Table (C);
8  end To_RGB;
9
10     function To_Int_Color (C : HTML_Color;
11                           S : HTML_RGB_Color) return Int_Color is
12     C_RGB : constant RGB := To_RGB (C);
13  begin
14      case S is
15          when Red    => return C_RGB.Red;
16          when Green  => return C_RGB.Green;
17          when Blue   => return C_RGB.Blue;
18      end case;
19  end To_Int_Color;
20
21     function Image (I : Int_Color) return String is
22         subtype Str_Range is Integer range 1 .. 10;
23         S : String (Str_Range);
24  begin
25     Ada.Integer_Text_IO.Put (To      => S,
26                             Item    => I,
27                             Base    => 16);
28     return S;
29  end Image;
30
31     function Image (C : RGB) return String is
32  begin
33         return ("(Red => "      & Image (C.Red)
34                 & ", Green => " & Image (C.Green)
35                 & ", Blue => "  & Image (C.Blue)
36                 & ")");
37  end Image;
38
39 end Color_Types;

```

Listing 155: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;       use Ada.Text_IO;
3
4  with Color_Types;      use Color_Types;
5
6  procedure Main is
7      type Test_Case_Index is
8          (HTML_Color_Red_Chk,
9           HTML_Color_Green_Chk,
10          HTML_Color_Blue_Chk);
11
12  procedure Check (TC : Test_Case_Index) is
13
14      procedure Check_HTML_Colors (S : HTML_RGB_Color) is
15      begin
16          Put_Line ("Selected: " & HTML_RGB_Color'Image (S));
17          for I in HTML_Color'Range loop

```

(continues on next page)

(continued from previous page)

```

18     Put_Line (HTML_Color'Image (I) & " => "
19             & Image (To_Int_Color (I, S)) & ".");
20   end loop;
21 end Check_HTML_Colors;

22
23 begin
24   case TC is
25     when HTML_Color_Red_Chk =>
26       Check_HTML_Colors (Red);
27     when HTML_Color_Green_Chk =>
28       Check_HTML_Colors (Green);
29     when HTML_Color_Blue_Chk =>
30       Check_HTML_Colors (Blue);
31   end case;
32 end Check;

33
34 begin
35   if Argument_Count < 1 then
36     Put_Line ("ERROR: missing arguments! Exiting...");
37     return;
38   elsif Argument_Count > 1 then
39     Put_Line ("Ignoring additional arguments...");
40   end if;
41
42   Check (Test_Case_Index'Value (Argument (1)));
43 end Main;

```

104.13 Object-oriented programming

104.13.1 Simple type extension

Listing 156: type_extensions.ads

```

1 package Type_Extensions is
2
3   type T_Float is tagged record
4     F : Float;
5   end record;
6
7   function Init (F : Float) return T_Float;
8
9   function Init (I : Integer) return T_Float;
10
11  function Image (T : T_Float) return String;
12
13  type T_Mixed is new T_Float with record
14    I : Integer;
15  end record;
16
17  function Init (F : Float) return T_Mixed;
18
19  function Init (I : Integer) return T_Mixed;
20
21  function Image (T : T_Mixed) return String;
22
23 end Type_Extensions;

```

Listing 157: type_extensions.adb

```

1 package body Type_Extensions is
2
3     function Init (F : Float) return T_Float is
4 begin
5     return ((F => F));
6 end Init;
7
8     function Init (I : Integer) return T_Float is
9 begin
10    return ((F => Float (I)));
11 end Init;
12
13     function Init (F : Float) return T_Mixed is
14 begin
15     return ((F => F,
16             I => Integer (F)));
17 end Init;
18
19     function Init (I : Integer) return T_Mixed is
20 begin
21     return ((F => Float (I),
22             I => I));
23 end Init;
24
25     function Image (T : T_Float) return String is
26 begin
27     return "{ F => " & Float'Image (T.F) & " }";
28 end Image;
29
30     function Image (T : T_Mixed) return String is
31 begin
32     return "{ F => " & Float'Image (T.F)
33             & ", I => " & Integer'Image (T.I) & " }";
34 end Image;
35
36 end Type_Extensions;

```

Listing 158: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Type_Extensions; use Type_Extensions;
5
6 procedure Main is
7
8     type Test_Case_Index is
9         (Type_Extension_Chk);
10
11     procedure Check (TC : Test_Case_Index) is
12         F1, F2 : T_Float;
13         M1, M2 : T_Mixed;
14     begin
15         case TC is
16             when Type_Extension_Chk =>
17                 F1 := Init (2.0);
18                 F2 := Init (3);
19                 M1 := Init (4.0);
20                 M2 := Init (5);

```

(continues on next page)

(continued from previous page)

```

21      if M2 in T_Float'Class then
22          Put_Line ("T_Mixed is in T_Float'Class as expected");
23      end if;
24
25
26          Put_Line ("F1: " & Image (F1));
27          Put_Line ("F2: " & Image (F2));
28          Put_Line ("M1: " & Image (M1));
29          Put_Line ("M2: " & Image (M2));
30      end case;
31  end Check;
32
33 begin
34     if Argument_Count < 1 then
35         Put_Line ("ERROR: missing arguments! Exiting...");
36         return;
37     elsif Argument_Count > 1 then
38         Put_Line ("Ignoring additional arguments...");
39     end if;
40
41     Check (Test_Case_Index'Value (Argument (1)));
42 end Main;

```

104.13.2 Online Store

Listing 159: online_store.ads

```

1  with Ada.Calendar; use Ada.Calendar;
2
3  package Online_Store is
4
5      type Amount is delta 10.0**(-2) digits 10;
6
7      subtype Percentage is Amount range 0.0 .. 1.0;
8
9      type Member is tagged record
10         Start : Year_Number;
11     end record;
12
13      type Member_Access is access Member'Class;
14
15      function Get_Status (M : Member) return String;
16
17      function Get_Price (M : Member;
18                          P : Amount) return Amount;
19
20      type Full_Member is new Member with record
21         Discount : Percentage;
22     end record;
23
24      function Get_Status (M : Full_Member) return String;
25
26      function Get_Price (M : Full_Member;
27                          P : Amount) return Amount;
28
29  end Online_Store;

```

Listing 160: online_store.adb

```

1 package body Online_Store is
2
3     function Get_Status (M : Member) return String is
4         ("Associate Member");
5
6     function Get_Status (M : Full_Member) return String is
7         ("Full Member");
8
9     function Get_Price (M : Member;
10                        P : Amount) return Amount is (P);
11
12    function Get_Price (M : Full_Member;
13                        P : Amount) return Amount is
14        (P * (1.0 - M.Discount));
15
16 end Online_Store;

```

Listing 161: online_store-tests.ads

```

1 package Online_Store.Tests is
2
3     procedure Simple_Test;
4
5 end Online_Store.Tests;

```

Listing 162: online_store-tests.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 package body Online_Store.Tests is
4
5     procedure Simple_Test is
6
7         type Member_Due_Amount is record
8             Member      : Member_Access;
9             Due_Amount : Amount;
10            end record;
11
12        function Get_Price (MA : Member_Due_Amount) return Amount is
13        begin
14            return MA.Member.Get_Price (MA.Due_Amount);
15        end Get_Price;
16
17        type Member_Due_Amounts is array (Positive range <>) of Member_Due_Amount;
18
19        DB : constant Member_Due_Amounts (1 .. 4)
20            := ((Member      => new Member'(Start => 2010),
21                  Due_Amount => 250.0),
22                  (Member      => new Full_Member'(Start      => 1998,
23                                              Discount => 0.1),
24                  Due_Amount => 160.0),
25                  (Member      => new Full_Member'(Start      => 1987,
26                                              Discount => 0.2),
27                  Due_Amount => 400.0),
28                  (Member      => new Member'(Start => 2013),
29                  Due_Amount => 110.0));
30
31        begin
32            for I in DB'Range loop
33                Put_Line ("Member #" & Positive'Image (I));

```

(continues on next page)

(continued from previous page)

```

33 Put_Line ("Status: " & DB (I).Member.Get_Status);
34 Put_Line ("Since: " & Year_Number'Image (DB (I).Member.Start));
35 Put_Line ("Due Amount: " & Amount'Image (Get_Price (DB (I)))); 
36 Put_Line ("-----");
37 end loop;
38 end Simple_Test;
39
40 end Online_Store.Tests;

```

Listing 163: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  with Online_Store;         use Online_Store;
5  with Online_Store.Tests;   use Online_Store.Tests;
6
7  procedure Main is
8
9    type Test_Case_Index is
10       (Type_Chk,
11        Unit_Test_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14
15    function Result_Image (Result : Boolean) return String is
16      (if Result then "OK" else "not OK");
17
18  begin
19    case TC is
20      when Type_Chk =>
21        declare
22          AM : constant Member      := (Start      => 2002);
23          FM : constant Full_Member := (Start      => 1990,
24                                         Discount => 0.2);
25        begin
26          Put_Line ("Testing Status of Associate Member Type => "
27                    & Result_Image (AM.Get_Status = "Associate Member"));
28          Put_Line ("Testing Status of Full Member Type => "
29                    & Result_Image (FM.Get_Status = "Full Member"));
30          Put_Line ("Testing Discount of Associate Member Type => "
31                    & Result_Image (AM.Get_Price (100.0) = 100.0));
32          Put_Line ("Testing Discount of Full Member Type => "
33                    & Result_Image (FM.Get_Price (100.0) = 80.0));
34        end;
35        when Unit_Test_Chk =>
36          Simple_Test;
37    end case;
38  end Check;
39
40 begin
41  if Argument_Count < 1 then
42    Put_Line ("ERROR: missing arguments! Exiting..."); 
43    return;
44  elsif Argument_Count > 1 then
45    Put_Line ("Ignoring additional arguments..."); 
46  end if;
47
48  Check (Test_Case_Index'Value (Argument (1)));
49 end Main;

```

104.14 Standard library: Containers

104.14.1 Simple todo list

Listing 164: todo_lists.ads

```

1  with Ada.Containers.Vectors;
2
3  package Todo_Lists is
4
5      type Todo_Item is access String;
6
7      package Todo_List_Pkg is new Ada.Containers.Vectors
8          (Index_Type    => Natural,
9           Element_Type => Todo_Item);
10
11     subtype Todo_List is Todo_List_Pkg.Vector;
12
13     procedure Add (Todos : in out Todo_List;
14                     Item   : String);
15
16     procedure Display (Todos : Todo_List);
17
18 end Todo_Lists;

```

Listing 165: todo_lists.adb

```

1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Todo_Lists is
4
5      procedure Add (Todos : in out Todo_List;
6                      Item   : String) is
7      begin
8          Todos.Append (new String'(Item));
9      end Add;
10
11     procedure Display (Todos : Todo_List) is
12     begin
13         Put_Line ("TO-DO LIST");
14         for T of Todos loop
15             Put_Line (T.all);
16         end loop;
17     end Display;
18
19 end Todo_Lists;

```

Listing 166: main.adb

```

1  with Ada.Command_Line;  use Ada.Command_Line;
2  with Ada.Text_IO;        use Ada.Text_IO;
3
4  with Todo_Lists;         use Todo_Lists;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Todo_List_Chk);
9
10     procedure Check (TC : Test_Case_Index) is
11         T : Todo_List;

```

(continues on next page)

(continued from previous page)

```

12 begin
13   case TC is
14     when Todo_List_Chk =>
15       Add (T, "Buy milk");
16       Add (T, "Buy tea");
17       Add (T, "Buy present");
18       Add (T, "Buy tickets");
19       Add (T, "Pay electricity bill");
20       Add (T, "Schedule dentist appointment");
21       Add (T, "Call sister");
22       Add (T, "Revise spreadsheet");
23       Add (T, "Edit entry page");
24       Add (T, "Select new design");
25       Add (T, "Create upgrade plan");
26       Display (T);
27   end case;
28 end Check;

29
30 begin
31   if Argument_Count < 1 then
32     Put_Line ("ERROR: missing arguments! Exiting...");
33     return;
34   elsif Argument_Count > 1 then
35     Put_Line ("Ignoring additional arguments...");
36   end if;
37
38   Check (Test_Case_Index'Value (Argument (1)));
39 end Main;

```

104.14.2 List of unique integers

Listing 167: ops.ads

```

1 with Ada.Containers.Ordered_Sets;
2
3 package Ops is
4
5   type Int_Array is array (Positive range <>) of Integer;
6
7   package Integer_Sets is new Ada.Containers.Ordered_Sets
8     (Element_Type => Integer);
9
10  subtype Int_Set is Integer_Sets.Set;
11
12  function Get_Unequal (A : Int_Array) return Int_Set;
13
14  function Get_Unequal (A : Int_Array) return Int_Array;
15
16 end Ops;

```

Listing 168: ops.adb

```

1 package body Ops is
2
3   function Get_Unequal (A : Int_Array) return Int_Set is
4     S : Int_Set;
5   begin
6     for E of A loop

```

(continues on next page)

(continued from previous page)

```

7      S.Include (E);
8  end loop;
9
10   return S;
11 end Get_Unique;
12
13 function Get_Unique (A : Int_Array) return Int_Array is
14   S : constant Int_Set := Get_Unique (A);
15   AR : Int_Array (1 .. Positive (S.Length));
16   I : Positive := 1;
17 begin
18   for E of S loop
19     AR (I) := E;
20     I := I + 1;
21   end loop;
22
23   return AR;
24 end Get_Unique;
25
26 end Ops;

```

Listing 169: main.adb

```

1 with Ada.Command_Line;           use Ada.Command_Line;
2 with Ada.Text_IO;               use Ada.Text_IO;
3
4 with Ops;                      use Ops;
5
6 procedure Main is
7   type Test_Case_Index is
8     (Get_Uneque_Set_Chk,
9      Get_Uneque_Array_Chk);
10
11 procedure Check (TC : Test_Case_Index;
12                   A : Int_Array) is
13
14   procedure Display_Uneque_Set (A : Int_Array) is
15     S : constant Int_Set := Get_Uneque (A);
16   begin
17     for E of S loop
18       Put_Line (Integer'Image (E));
19     end loop;
20   end Display_Uneque_Set;
21
22   procedure Display_Uneque_Array (A : Int_Array) is
23     AU : constant Int_Array := Get_Uneque (A);
24   begin
25     for E of AU loop
26       Put_Line (Integer'Image (E));
27     end loop;
28   end Display_Uneque_Array;
29
30 begin
31   case TC is
32     when Get_Uneque_Set_Chk => Display_Uneque_Set (A);
33     when Get_Uneque_Array_Chk => Display_Uneque_Array (A);
34   end case;
35 end Check;
36
37 begin
38   if Argument_Count < 3 then

```

(continues on next page)

(continued from previous page)

```

39      Put_Line ("ERROR: missing arguments! Exiting...");  

40      return;  

41  else  

42    declare  

43      A : Int_Array (1 .. Argument_Count - 1);  

44    begin  

45      for I in A'Range loop  

46        A (I) := Integer'Value (Argument (1 + I));  

47      end loop;  

48      Check (Test_Case_Index'Value (Argument (1)), A);  

49    end;  

50  end if;  

51 end Main;

```

104.15 Standard library: Dates & Times

104.15.1 Holocene calendar

Listing 170: to_holocene_year.adb

```

1  with Ada.Calendar; use Ada.Calendar;  

2  

3  function To_Holocene_Year (T : Time) return Integer is  

4  begin  

5    return Year (T) + 10_000;  

6  end To_Holocene_Year;

```

Listing 171: main.adb

```

1  with Ada.Command_Line;           use Ada.Command_Line;  

2  with Ada.Text_IO;               use Ada.Text_IO;  

3  with Ada.Calendar;              use Ada.Calendar;  

4  

5  with To_Holocene_Year;  

6  

7  procedure Main is  

8    type Test_Case_Index is  

9      (Holocene_Chk);  

10  

11   procedure Display_Holocene_Year (Y : Year_Number) is  

12     HY : Integer;  

13   begin  

14     HY := To_Holocene_Year (Time_Of (Y, 1, 1));  

15     Put_Line ("Year (Gregorian): " & Year_Number'Image (Y));  

16     Put_Line ("Year (Holocene): " & Integer'Image (HY));  

17   end Display_Holocene_Year;  

18  

19   procedure Check (TC : Test_Case_Index) is  

20   begin  

21     case TC is  

22       when Holocene_Chk =>  

23         Display_Holocene_Year (2012);  

24         Display_Holocene_Year (2020);  

25     end case;  

26   end Check;

```

(continues on next page)

(continued from previous page)

```

28 begin
29   if Argument_Count < 1 then
30     Put_Line ("ERROR: missing arguments! Exiting...");  

31     return;
32   elsif Argument_Count > 1 then
33     Put_Line ("Ignoring additional arguments...");  

34   end if;  

35  

36   Check (Test_Case_Index'Value (Argument (1)));
37 end Main;

```

104.15.2 List of events

Listing 172: events.ads

```

1 with Ada.Containers.Vectors;
2
3 package Events is
4
5   type Event_Item is access String;
6
7   package Event_Item_Containers is new
8     Ada.Containers.Vectors
9       (Index_Type    => Positive,
10        Element_Type => Event_Item);
11
12  subtype Event_Items is Event_Item_Containers.Vector;
13
14 end Events;

```

Listing 173: events-lists.ads

```

1 with Ada.Calendar;           use Ada.Calendar;
2 with Ada.Containers.Ordered_Maps;
3
4 package Events.Lists is
5
6   type Event_List is tagged private;
7
8   procedure Add (Events      : in out Event_List;
9                  Event_Time :          Time;
10                 Event      :          String);
11
12  procedure Display (Events : Event_List);
13
14 private
15
16  package Event_Time_Item_Containers is new
17    Ada.Containers.Ordered_Maps
18      (Key_Type      => Time,
19       Element_Type  => Event_Items,
20        "="          => Event_Item_Containers."=");
21
22  type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;

```

Listing 174: events-lists.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4  package body Events.Lists is
5
6    procedure Add (Events      : in out Event_List;
7                  Event_Time : Time;
8                  Event      : String) is
9      use Event_Item_Containers;
10     E : constant Event_Item := new String'(Event);
11 begin
12   if not Events.Contains (Event_Time) then
13     Events.Include (Event_Time, Empty_Vector);
14   end if;
15   Events (Event_Time).Append (E);
16 end Add;
17
18 function Date_Image (T : Time) return String is
19   Date_Img : constant String := Image (T);
20 begin
21   return Date_Img (1 .. 10);
22 end;
23
24 procedure Display (Events : Event_List) is
25   use Event_Time_Item_Containers;
26   T : Time;
27 begin
28   Put_Line ("EVENTS LIST");
29   for C in Events.Iterate loop
30     T := Key (C);
31     Put_Line ("- " & Date_Image (T));
32     for I of Events (C) loop
33       Put_Line ("      - " & I.all);
34     end loop;
35   end loop;
36 end Display;
37
38 end Events.Lists;

```

Listing 175: main.adb

```

1  with Ada.Command_Line;           use Ada.Command_Line;
2  with Ada.Text_IO;               use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
5
6  with Events.Lists;             use Events.Lists;
7
8  procedure Main is
9    type Test_Case_Index is
10      (Event_List_Chk);
11
12  procedure Check (TC : Test_Case_Index) is
13    EL : Event_List;
14  begin
15    case TC is
16      when Event_List_Chk =>
17        EL.Add (Time_Of (2018, 2, 16),
18                 "Final check");

```

(continues on next page)

(continued from previous page)

```

19      EL.Add (Time_Of (2018, 2, 16),
20                "Release");
21      EL.Add (Time_Of (2018, 12, 3),
22                "Brother's birthday");
23      EL.Add (Time_Of (2018, 1, 1),
24                "New Year's Day");
25      EL.Display;
26  end case;
27 end Check;

28
29 begin
30  if Argument_Count < 1 then
31    Put_Line ("ERROR: missing arguments! Exiting...");  

32    return;
33  elsif Argument_Count > 1 then
34    Put_Line ("Ignoring additional arguments...");  

35  end if;
36
37  Check (Test_Case_Index'Value (Argument (1)));
38 end Main;

```

104.16 Standard library: Strings

104.16.1 Concatenation

Listing 176: str_concat.ads

```

1  with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
2
3  package Str_Concat is
4
5    type Unbounded_Strings is array (Positive range <>) of Unbounded_String;
6
7    function Concat (USA          : Unbounded_Strings;
8                      Trim_Str     : Boolean;
9                      Add_Whitespace : Boolean) return Unbounded_String;
10
11   function Concat (USA          : Unbounded_Strings;
12                      Trim_Str     : Boolean;
13                      Add_Whitespace : Boolean) return String;
14
15 end Str_Concat;

```

Listing 177: str_concat.adb

```

1  with Ada.Strings; use Ada.Strings;
2
3  package body Str_Concat is
4
5    function Concat (USA          : Unbounded_Strings;
6                      Trim_Str     : Boolean;
7                      Add_Whitespace : Boolean) return Unbounded_String is
8
9      function Retrieve (USA          : Unbounded_Strings;
10                         Trim_Str     : Boolean;
11                         Index        : Positive) return Unbounded_String is

```

(continues on next page)

(continued from previous page)

```

12      US_Internal : Unbounded_String := USA (Index);
13  begin
14    if Trim_Str then
15      US_Internal := Trim (US_Internal, Both);
16    end if;
17    return US_Internal;
18  end Retrieve;
19
20  US : Unbounded_String := To_Unbounded_String ("");
21  begin
22    for I in USA'First .. USA'Last - 1 loop
23      US := US & Retrieve (USA, Trim_Str, I);
24      if Add_Whitespace then
25        US := US & " ";
26      end if;
27    end loop;
28    US := US & Retrieve (USA, Trim_Str, USA'Last);
29
30    return US;
31  end Concat;
32
33  function Concat (USA           : Unbounded_Strings;
34                    Trim_Str       : Boolean;
35                    Add_Whitespace : Boolean) return String is
36  begin
37    return To_String (Concat (USA, Trim_Str, Add_Whitespace));
38  end Concat;
39
40 end Str_Concat;

```

Listing 178: main.adb

```

1  with Ada.Command_Line;          use Ada.Command_Line;
2  with Ada.Text_IO;              use Ada.Text_IO;
3  with Ada.Strings.Unbounded;    use Ada.Strings.Unbounded;
4
5  with Str_Concat;              use Str_Concat;
6
7  procedure Main is
8    type Test_Case_Index is
9      (Unbounded_Concat_No_Trim_No_WS_Chk,
10       Unbounded_Concat_Trim_No_WS_Chk,
11       String_Concat_Trim_WS_Chk,
12       Concat_Single_Element);
13
14  procedure Check (TC : Test_Case_Index) is
15  begin
16    case TC is
17      when Unbounded_Concat_No_Trim_No_WS_Chk =>
18        declare
19          S : constant Unbounded_Strings := (
20            To_Unbounded_String ("Hello"),
21            To_Unbounded_String (" World"),
22            To_Unbounded_String ("!"));
23        begin
24          Put_Line (To_String (Concat (S, False, False)));
25        end;
26      when Unbounded_Concat_Trim_No_WS_Chk =>
27        declare
28          S : constant Unbounded_Strings := (
29            To_Unbounded_String (" This "));

```

(continues on next page)

(continued from previous page)

```

30      To_Unbounded_String (" _is_ "),
31      To_Unbounded_String (" a "),
32      To_Unbounded_String (" _check "));
33  begin
34      Put_Line (To_String (Concat (S, True, False)));
35  end;
36  when String_Concat_Trim_WS_Chk =>
37  declare
38      S : constant Unbounded.Strings := (
39          To_Unbounded_String (" This "),
40          To_Unbounded_String (" is a "),
41          To_Unbounded_String (" test. "));
42  begin
43      Put_Line (Concat (S, True, True));
44  end;
45  when Concat_Single_Element =>
46  declare
47      S : constant Unbounded.Strings := (
48          1 => To_Unbounded_String (" Hi "));
49  begin
50      Put_Line (Concat (S, True, True));
51  end;
52  end case;
53 end Check;

54
55 begin
56  if Argument_Count < 1 then
57      Put_Line ("ERROR: missing arguments! Exiting... ");
58      return;
59  elsif Argument_Count > 1 then
60      Put_Line ("Ignoring additional arguments... ");
61  end if;
62
63  Check (Test_Case_Index'Value (Argument (1)));
64 end Main;

```

104.16.2 List of events

Listing 179: events.ads

```

1  with Ada.Strings.Unbounded;  use Ada.Strings.Unbounded;
2  with Ada.Containers.Vectors;
3
4  package Events is
5
6      subtype Event_Item is Unbounded_String;
7
8      package Event_ItemContainers is new
9          Ada.Containers.Vectors
10         (Index_Type    => Positive,
11          Element_Type => Event_Item);
12
13      subtype Event_Items is Event_ItemContainers.Vector;
14
15 end Events;

```

Listing 180: events-lists.ads

```

1  with Ada.Calendar;           use Ada.Calendar;
2  with Ada.Containers.Ordered_Maps;
3
4  package Events.Lists is
5
6    type Event_List is tagged private;
7
8    procedure Add (Events      : in out Event_List;
9                  Event_Time :        Time;
10                 Event      :        String);
11
12   procedure Display (Events : Event_List);
13
14  private
15
16  package Event_Time_Item_Containers is new
17    Ada.Containers.Ordered_Maps
18    (Key_Type      => Time,
19     Element_Type  => Event_Items,
20     "="          => Event_Item_Containers."=");
21
22  type Event_List is new Event_Time_Item_Containers.Map with null record;
23
24 end Events.Lists;

```

Listing 181: events-lists.adb

```

1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Calendar.Formatting; use Ada.Calendar.Formatting;
3
4  package body Events.Lists is
5
6    procedure Add (Events      : in out Event_List;
7                  Event_Time : Time;
8                  Event      : String) is
9      use Event_Item_Containers;
10     E : constant Event_Item := To_Unbounded_String (Event);
11
12 begin
13   if not Events.Contains (Event_Time) then
14     Events.Include (Event_Time, Empty_Vector);
15   end if;
16   Events (Event_Time).Append (E);
17 end Add;
18
19 function Date_Image (T : Time) return String is
20   Date_Img : constant String := Image (T);
21
22 begin
23   return Date_Img (1 .. 10);
24 end;
25
26
27 procedure Display (Events : Event_List) is
28   use Event_Time_Item_Containers;
29   T : Time;
30
31 begin
32   Put_Line ("EVENTS LIST");
33   for C in Events.Iterate loop
34     T := Key (C);
35     Put_Line ("- " & Date_Image (T));
36     for I of Events (C) loop

```

(continues on next page)

(continued from previous page)

```

33      Put_Line ("      - " & To_String (I));
34   end loop;
35 end loop;
36 end Display;
37
38 end Events.Lists;

```

Listing 182: main.adb

```

1  with Ada.Command_Line;           use Ada.Command_Line;
2  with Ada.Text_IO;               use Ada.Text_IO;
3  with Ada.Calendar;
4  with Ada.Calendar.Formatting;   use Ada.Calendar.Formatting;
5  with Ada.Strings.Unbounded;     use Ada.Strings.Unbounded;
6
7  with Events;
8  with Events.Lists;             use Events.Lists;
9
10 procedure Main is
11   type Test_Case_Index is
12     (Unbounded_String_Chk,
13      Event_List_Chk);
14
15   procedure Check (TC : Test_Case_Index) is
16     EL : Event_List;
17   begin
18     case TC is
19       when Unbounded_String_Chk =>
20         declare
21           S : constant Events.Event_Item := To_Unbounded_String ("Checked");
22         begin
23           Put_Line (To_String (S));
24         end;
25       when Event_List_Chk =>
26         EL.Add (Time_Of (2018, 2, 16),
27                 "Final check");
28         EL.Add (Time_Of (2018, 2, 16),
29                 "Release");
30         EL.Add (Time_Of (2018, 12, 3),
31                 "Brother's birthday");
32         EL.Add (Time_Of (2018, 1, 1),
33                 "New Year's Day");
34         EL.Display;
35     end case;
36   end Check;
37
38 begin
39   if Argument_Count < 1 then
40     Put_Line ("ERROR: missing arguments! Exiting...");
41     return;
42   elsif Argument_Count > 1 then
43     Put_Line ("Ignoring additional arguments...");
44   end if;
45
46   Check (Test_Case_Index'Value (Argument (1)));
47 end Main;

```

104.17 Standard library: Numerics

104.17.1 Decibel Factor

Listing 183: decibels.ads

```

1 package Decibels is
2
3     subtype Decibel is Float;
4     subtype Factor   is Float;
5
6     function To_Decibel (F : Factor) return Decibel;
7
8     function To_Factor (D : Decibel) return Factor;
9
10 end Decibels;

```

Listing 184: decibels.adb

```

1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Decibels is
4
5     function To_Decibel (F : Factor) return Decibel is
6     begin
7         return 20.0 * Log (F, 10.0);
8     end To_Decibel;
9
10    function To_Factor (D : Decibel) return Factor is
11    begin
12        return 10.0 ** (D / 20.0);
13    end To_Factor;
14
15 end Decibels;

```

Listing 185: main.adb

```

1 with Ada.Command_Line; use Ada.Command_Line;
2 with Ada.Text_IO;      use Ada.Text_IO;
3
4 with Decibels;          use Decibels;
5
6 procedure Main is
7     type Test_Case_Index is
8         (Db_Chk,
9          Factor_Chk);
10
11    procedure Check (TC : Test_Case_Index; V : Float) is
12
13        package F_IO is new Ada.Text_IO.Float_IO (Factor);
14        package D_IO is new Ada.Text_IO.Float_IO (Decibel);
15
16        procedure Put_Decibel_Cnvt (D : Decibel) is
17            F : constant Factor := To_Factor (D);
18        begin
19            D_IO.Put (D, 0, 2, 0);
20            Put (" dB => Factor of ");
21            F_IO.Put (F, 0, 2, 0);
22            New_Line;
23        end;

```

(continues on next page)

(continued from previous page)

```

24
25      procedure Put_Factor_Cnvt (F : Factor) is
26          D : constant Decibel := To_Decibel (F);
27      begin
28          Put ("Factor of ");
29          F_I0.Put (F, 0, 2, 0);
30          Put (" => ");
31          D_I0.Put (D, 0, 2, 0);
32          Put_Line (" dB");
33      end;
34      begin
35          case TC is
36              when Db_Chk =>
37                  Put_Decibel_Cnvt (Decibel (V));
38              when Factor_Chk =>
39                  Put_Factor_Cnvt (Factor (V));
40          end case;
41      end Check;
42
43  begin
44      if Argument_Count < 2 then
45          Put_Line ("ERROR: missing arguments! Exiting...");
46          return;
47      elsif Argument_Count > 2 then
48          Put_Line ("Ignoring additional arguments...");
49      end if;
50
51      Check (Test_Case_Index'Value (Argument (1)), Float'Value (Argument (2)));
52  end Main;

```

104.17.2 Root-Mean-Square

Listing 186: signals.ads

```

1 package Signals is
2
3     subtype Sig_Value is Float;
4
5     type Signal is array (Natural range <>) of Sig_Value;
6
7     function Rms (S : Signal) return Sig_Value;
8
9 end Signals;

```

Listing 187: signals.adb

```

1 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
2
3 package body Signals is
4
5     function Rms (S : Signal) return Sig_Value is
6         Acc : Float := 0.0;
7     begin
8         for V of S loop
9             Acc := Acc + V * V;
10        end loop;
11
12        return Sqrt (Acc / Float (S'Length));

```

(continues on next page)

(continued from previous page)

```

13   end;
14
15 end Signals;
```

Listing 188: signals-std.ads

```

1 package Signals.Std is
2
3   Sample_Rate : Float := 8000.0;
4
5   function Generate_Sine (N : Positive; Freq : Float) return Signal;
6
7   function Generate_Square (N : Positive) return Signal;
8
9   function Generate_Triangular (N : Positive) return Signal;
10
11 end Signals.Std;
```

Listing 189: signals-std.adb

```

1 with Ada.Numerics;           use Ada.Numerics;
2 with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
3
4 package body Signals.Std is
5
6   function Generate_Sine (N : Positive; Freq : Float) return Signal is
7     S : Signal (0 .. N - 1);
8   begin
9     for I in S'First .. S'Last loop
10       S (I) := 1.0 * Sin (2.0 * Pi * (Freq * Float (I) / Sample_Rate));
11     end loop;
12
13   return S;
14 end;
15
16   function Generate_Square (N : Positive) return Signal is
17     S : constant Signal (0 .. N - 1) := (others => 1.0);
18   begin
19     return S;
20   end;
21
22   function Generate_Triangular (N : Positive) return Signal is
23     S      : Signal (0 .. N - 1);
24     S_Half : constant Natural := S'Last / 2;
25   begin
26     for I in S'First .. S_Half loop
27       S (I) := 1.0 * (Float (I) / Float (S_Half));
28     end loop;
29     for I in S_Half .. S'Last loop
30       S (I) := 1.0 - (1.0 * (Float (I - S_Half) / Float (S_Half)));
31     end loop;
32
33   return S;
34 end;
35
36 end Signals.Std;
```

Listing 190: main.adb

```

1  with Ada.Command_Line;      use Ada.Command_Line;
2  with Ada.Text_IO;          use Ada.Text_IO;
3
4  with Signals;              use Signals;
5  with Signals.Std;          use Signals.Std;
6
7  procedure Main is
8    type Test_Case_Index is
9      (Sine_Signal_Chk,
10       Square_Signal_Chk,
11       Triangular_Signal_Chk);
12
13  procedure Check (TC : Test_Case_Index) is
14    package Sig_IO is new Ada.Text_IO.Float_IO (Sig_Value);
15
16    N      : constant Positive := 1024;
17    S_Si   : constant Signal := Generate_Sine (N, 440.0);
18    S_Sq   : constant Signal := Generate_Square (N);
19    S_Tr   : constant Signal := Generate_Triangular (N + 1);
20
21 begin
22   case TC is
23     when Sine_Signal_Chk =>
24       Put ("RMS of Sine Signal: ");
25       Sig_IO.Put (Rms (S_Si), 0, 2, 0);
26       New_Line;
27     when Square_Signal_Chk =>
28       Put ("RMS of Square Signal: ");
29       Sig_IO.Put (Rms (S_Sq), 0, 2, 0);
30       New_Line;
31     when Triangular_Signal_Chk =>
32       Put ("RMS of Triangular Signal: ");
33       Sig_IO.Put (Rms (S_Tr), 0, 2, 0);
34       New_Line;
35   end case;
36 end Check;
37
38 begin
39   if Argument_Count < 1 then
40     Put_Line ("ERROR: missing arguments! Exiting... ");
41     return;
42   elsif Argument_Count > 1 then
43     Put_Line ("Ignoring additional arguments... ");
44   end if;
45
46   Check (Test_Case_Index'Value (Argument (1)));
end Main;

```

104.17.3 Rotation

Listing 191: rotation.ads

```

1  with Ada.Numerics.Complex_Types;
2  use Ada.Numerics.Complex_Types;
3
4  package Rotation is
5
6    type Complex_Points is array (Positive range <>) of Complex;

```

(continues on next page)

(continued from previous page)

```

7   function Rotation (N : Positive) return Complex_Points;
8
9 end Rotation;
```

Listing 192: rotation.adb

```

1  with Ada.Numerics; use Ada.Numerics;
2
3 package body Rotation is
4
5   function Rotation (N : Positive) return Complex_Points is
6     C_Angle : constant Complex := Compose_From_Polar (1.0, 2.0 * Pi / Float (N));
7
8 begin
9   return C : Complex_Points (1 .. N + 1) do
10    C (1) := Compose_From_Cartesian (1.0, 0.0);
11
12    for I in C'First + 1 .. C'Last loop
13      C (I) := C (I - 1) * C_Angle;
14    end loop;
15  end return;
16 end;
17
18 end Rotation;
```

Listing 193: angles.ads

```

1  with Rotation; use Rotation;
2
3 package Angles is
4
5   subtype Angle is Float;
6
7   type Angles is array (Positive range <>) of Angle;
8
9   function To_Angles (C : Complex_Points) return Angles;
10
11 end Angles;
```

Listing 194: angles.adb

```

1  with Ada.Numerics;           use Ada.Numerics;
2  with Ada.Numerics.Complex_Types; use Ada.Numerics.Complex_Types;
3
4 package body Angles is
5
6   function To_Angles (C : Complex_Points) return Angles is
7 begin
8   return A : Angles (C'Range) do
9     for I in A'Range loop
10      A (I) := Argument (C (I)) / Pi * 180.0;
11    end loop;
12  end return;
13 end To_Angles;
14
15 end Angles;
```

Listing 195: rotation-tests.ads

```

1 package Rotation.Tests is
2
3   procedure Test_Rotation (N : Positive);
4
5   procedure Test_Angles (N : Positive);
6
7 end Rotation.Tests;

```

Listing 196: rotation-tests.adb

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Text_IO.Complex_IO; use Ada.Numerics;
3 with Ada.Numerics;          use Ada.Numerics;
4
5 with Angles;               use Angles;
6
7 package body Rotation.Tests is
8
9   package C_IO is new Ada.Text_IO.Complex_IO (Complex_Types);
10  package F_IO is new Ada.Text_IO.Float_IO (Float);
11
12  --
13  -- Adapt value due to floating-point inaccuracies
14  --
15
16  function Adapt (C : Complex) return Complex is
17    function Check_Zero (F : Float) return Float is
18      (if F <= 0.0 and F >= -0.01 then 0.0 else F);
19
20 begin
21   return C_Out : Complex := C do
22     C_Out.Re := Check_Zero (C_Out.Re);
23     C_Out.Im := Check_Zero (C_Out.Im);
24   end return;
25 end Adapt;
26
27 function Adapt (A : Angle) return Angle is
28   (if A <= -179.99 and A >= -180.01 then 180.0 else A);
29
30 procedure Test_Rotation (N : Positive) is
31   C : constant Complex_Points := Rotation (N);
32
33 begin
34   Put_Line ("---- Points for " & Positive'Image (N) & " slices ----");
35   for V of C loop
36     Put ("Point: ");
37     C_IO.Put (Adapt (V), 0, 1, 0);
38     New_Line;
39   end loop;
40 end Test_Rotation;
41
42 procedure Test_Angles (N : Positive) is
43   C : constant Complex_Points := Rotation (N);
44   A : constant Angles.Angles := To_Angles (C);
45
46 begin
47   Put_Line ("---- Angles for " & Positive'Image (N) & " slices ----");
48   for V of A loop
49     Put ("Angle: ");
      F_IO.Put (Adapt (V), 0, 2, 0);
      Put_Line (" degrees");
    end loop;

```

(continues on next page)

(continued from previous page)

```
50    end Test_Angles;
51
52 end Rotation.Tests;
```

Listing 197: main.adb

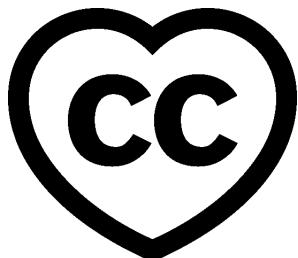
```
1  with Ada.Command_Line;          use Ada.Command_Line;
2  with Ada.Text_IO;              use Ada.Text_IO;
3
4  with Rotation.Tests;          use Rotation.Tests;
5
6  procedure Main is
7      type Test_Case_Index is
8          (Rotation_Chk,
9           Angles_Chk);
10
11     procedure Check (TC : Test_Case_Index; N : Positive) is
12     begin
13         case TC is
14             when Rotation_Chk =>
15                 Test_Rotation (N);
16             when Angles_Chk =>
17                 Test_Angles (N);
18         end case;
19     end Check;
20
21 begin
22     if Argument_Count < 2 then
23         Put_Line ("ERROR: missing arguments! Exiting...");
24         return;
25     elsif Argument_Count > 2 then
26         Put_Line ("Ignoring additional arguments...");
27     end if;
28
29     Check (Test_Case_Index'Value (Argument (1)), Positive'Value (Argument (2)));
30 end Main;
```

Part X

Bug Free Coding with SPARK Ada

Copyright © 2018 – 2022, AdaCore

This book is published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details [on this page¹⁴⁰](#)



Workshop project: Learn to write maintainable bug-free code with SPARK Ada.

This document was written by Robert Tice.

¹⁴⁰ <http://creativecommons.org/licenses/by-sa/4.0>

LET'S BUILD A STACK

In this lab we will build a stack data structure and use the SPARK provers to find the errors in the below implementation.

105.1 Background

So, what is a stack?

A stack is like a pile of dishes...



1. The pile starts out empty.
2. You add (push) a new plate (data) to the stack by placing it on the top of the pile.
3. To get plates (data) out, you take the one off the top of the pile (pop).
4. Our stack has a maximum height (size) of 9 dishes

Pushing items onto the stack

Here's what should happen if we pushed the string MLH onto the stack.

Step 0:
Empty

1:
2:
3:
4:
5:

Last = 0

Step 1:
Push('M')

1: M
2:
3:
4:
5:

Last = 1

Step 2:
Push('L')

1: M
2: L
3:
4:
5:

Last = 2

Step 3:
Push('H')

1: M
2: L
3: H
4:
5:

Last = 3

Step 4:
Top()

1: M
2: L
3: H
4:
5:

Last = 3

returns:

'H'

The list starts out empty. Each time we push a character onto the stack, Last increments by 1.

Popping items from the stack

Here's what should happen if we popped 2 characters off our stack & then clear it.

Step 0:

Start

1:	M
2:	L
3:	H
4:	
5:	

Last = 3

Step 1:

Pop()

1:	M
2:	L
3:	H
4:	
5:	

Last = 2

returns:

'H'

Step 2:

Pop()

1:	M
2:	L
3:	H
4:	
5:	

Last = 1

returns:

'L'

Step 3:

Clear()

1:	M
2:	L
3:	H
4:	
5:	

Last = 0

Note that pop and clear don't unset the Storage array's elements, they just change the value of Last.

105.2 Input Format

N inputs will be read from stdin/console as inputs, C to the stack.

105.3 Constraints

1 <= N <= 1000

C is any character. Characters d and p will be special characters corresponding to the below commands:

p => Pops a character off the stack

d => Prints the current characters in the stack

105.4 Output Format

If the stack currently has the characters "M", "L", and "H" then the program should print the stack like this:

[M, L, H]

105.5 Sample Input

M L H d p d p d p d

105.6 Sample Output

[M, L, H] [M, L] [M] []

Listing 1: stack.ads

```
1 package Stack with SPARK_Mode => On is
2
3     procedure Push (V : Character)
4         with Pre => not Full,
5              Post => Size = Size'Old + 1;
6
7     procedure Pop (V : out Character)
8         with Pre => not Empty,
9              Post => Size = Size'Old - 1;
10
11    procedure Clear
12        with Post => Size = 0;
13
14    function Top return Character
15        with Post => Top'Result = Tab>Last;
16
17    Max_Size : constant := 9;
```

(continues on next page)

(continued from previous page)

```

18  -- The stack size.
19
20 Last : Integer range 0 .. Max_Size := 0;
21  -- Indicates the top of the stack. When 0 the stack is empty.
22
23 Tab : array (1 .. Max_Size) of Character;
24  -- The stack. We push and pop pointers to Values.
25
26 function Full return Boolean is (Last = Max_Size);
27
28 function Empty return Boolean is (Last < 1);
29
30 function Size return Integer is (Last);
31
32 end Stack;

```

Listing 2: stack.adb

```

1 package body Stack with SPARK_Mode => On is
2
3  -----
4  -- Clear --
5  -----
6
7  procedure Clear
8  is
9  begin
10    Last := Tab'First;
11  end Clear;
12
13  -----
14  -- Push --
15  -----
16
17  procedure Push (V : Character)
18  is
19  begin
20    Tab (Last) := V;
21  end Push;
22
23  -----
24  -- Pop --
25  -----
26
27  procedure Pop (V : out Character)
28  is
29  begin
30    Last := Last - 1;
31    V := Tab (Last);
32  end Pop;
33
34  -----
35  -- Top --
36  -----
37
38  function Top return Character
39  is
40  begin
41    return Tab (1);
42  end Top;
43

```

(continues on next page)

44 **end** Stack;

Listing 3: main.adb

```

1  with Ada.Command_Line; use Ada.Command_Line;
2  with Ada.Text_IO;      use Ada.Text_IO;
3  with Stack;           use Stack;
4
5  procedure Main with SPARK_Mode => Off
6  is
7
8      -----
9      -- Debug --
10     -----
11
12     procedure Debug
13     is
14     begin
15
16         if not Stack.Empty then
17
18             Put ("[");
19             for I in Stack.Tab'First .. Stack.Size - 1 loop
20                 Put (Stack.Tab (I) & ", ");
21             end loop;
22             Put_Line (Stack.Tab (Stack.Size) & "]");
23         else
24             Put_Line ("[]");
25         end if;
26
27     end Debug;
28
29     S : Character;
30
31 begin
32
33     -----
34     -- Main --
35     -----
36
37     for Arg in 1 .. Argument_Count loop
38         if Argument (Arg)'Length /= 1 then
39             Put_Line (Argument (Arg) & " is an invalid input to the stack.");
40         else
41             S := Argument (Arg)(Argument (Arg)'First);
42
43             if S = 'd' then
44                 Debug;
45             elsif S = 'p' then
46                 if not Stack.Empty then
47                     Stack.Pop (S);
48                 else
49                     Put_Line ("Nothing to Pop, Stack is empty!");
50                 end if;
51             else
52                 if not Stack.Full then
53                     Stack.Push (S);
54                 else
55                     Put_Line ("Could not push '" & S & "'", Stack is full!");
56                 end if;
57             end if;

```

(continues on next page)

(continued from previous page)

```
58      end if;  
59  
60  end loop;  
61  
62 end Main;
```

BIBLIOGRAPHY

[Jorvik] 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.