# A Brief Overview of Ada and SPARK

## All the Essentials, None of the Overwhelm

# *Core* Language Content

- Ada is a **compiled**, **multi-paradigm** language
  - Exceptions
  - Generic units
  - Dynamic memory management
  - Low-level programming
  - Object-Oriented Programming (OOP)
  - Concurrent programming
  - Contract-Based Programming
- With a **static** and **strong** type model

# Declarations

# Identifiers

- Syntax

```
identifier ::= letter {[underline] letter_or_digit}
```

- Character set **Unicode** 4.0
  - 8, 16, 32 bit-wide characters
- Case **not significant**
  - **SpacePerson** ⇔ **SPACEPERSON**
  - but **different** from **Space_Person**
- Reserved words are **forbidden**

# Comments

- Terminate at end of line (i.e., no comment terminator sequence)

```
-- This is a multi-
-- line comment
A : B; -- this is an end-of-line comment
```

# Decimal Numeric Literals

- Syntax

```
decimal_literal ::=
  numeral [.num] E [+numeral|-numeral]
numeral ::= digit {[underline] digit}
```

- Underscore is not significant
- **E** (exponent) must always be integer
- Examples

```
12      0      1E6         123_456
12.0    0.0    3.14159_26  2.3E-4
```

# Based Numeric Literals

```
based_literal ::= base # numeral [.numeral] #
exponent
numeral ::= base_digit { '_' base_digit }
```

- Base can be 2 .. 16
- Exponent is always a base 10 integer

```
16#FFF#           => 4095
2#1111_1111_1111# => 4095 -- With underline
16#F.FF#E+2       => 4095.0
8#10#E+3          => 4096 (8 * 8**3)
```

# Object Declarations

- Variables and constants
- Basic Syntax

```
<name> : subtype_indication [:= <initial value>];
```

- Examples

```
Z, Phase : Analog;
Max : constant Integer := 200;
-- variable with a constraint
Count : Integer range 0 .. Max := 0;
-- dynamic initial value via function call
Root : Tree := F(X);
-- Will call G(X) twice, once per variable
A, B : Integer := G(X);
```

# Basic Types

# Ada Type Model

- *Static* Typing
  - Object type **cannot change**
- *Strong* Typing
  - By **name**
  - **Compiler-enforced** operations and values
  - **Explicit** conversion for “related” types
  - **Unchecked** conversions possible

# Ada “Named Typing”

- **Name** differentiate types
- Structure does **not**
- Identical structures may **not** be interoperable

```
type Yen is range 0 .. 100_000_000;
type Kilometers is range 0 .. 100_000_000;
Money : Yen;
Distance : Kilometers;
...
Money := Distance; -- not legal
```

# Attributes

- Functions *associated* with a type
  - May take input parameters
- Some are language-defined
  - *May* be implementation-defined
  - **Built-in**
  - Cannot be user-defined
  - Some can be overridden
- Examples

```
Typemark'Size
Integer'Max (A, B);
```

# Numeric Types

# Signed Integer Types

- Range of signed **whole** numbers
  - Symmetric about zero (-0 = +0)
- Syntax

```
type <identifier> is range  <lower> .. <upper>;
```

- Implicit numeric operators

```
-- 12-bit device
type Analog_Conversions is range 0 .. 4095;
Count : Analog_Conversions;
...
begin
   ...
   Count := Count + 1;
   ...
end;
```

# Range Attributes For All Scalars

- `T'First`
  - First (**smallest**) value of type `T`
- `T'Last`
  - Last (**greatest**) value of type `T`
- `T'Range`
  - Shorthand for `T'First .. T'Last`

```
type Signed_T is range -99 .. 100;
Smallest : Signed_T := Signed_T'First; -- -99
Largest  : Signed_T := Signed_T'Last;  -- 100
```

# Declaring Floating Point Types

- Syntax

```
type <identifier> is
     digits <expression> [range constraint];
```

  - *digits* → **minimum** number of significant digits
  - **Decimal** digits, not bits
- Compiler choses representation
  - From **available** floating point types
  - May be **more** accurate, but not less
  - If none available → declaration is **rejected**

# Enumeration Types

# Enumeration Types

- Enumeration of **logical** values
  - Integer value is an implementation detail
- Syntax

```
type <identifier> is (<identifier-list>) ;
```

- Literals
  - Distinct, ordered
  - Can be in **multiple** enumerations

```
type Colors is (Red, Orange, Yellow, Green, Blue, Violet);
type Stop_Light is (Red, Yellow, Green);
...
-- Red a member of both Colors and Stop_Light
Shade : Colors := Red;
Light : Stop_Light := Red;
```

# Enumeration Type Operations

- Assignment, relationals
- **Not** numeric quantities
  - *Possible* with attributes
  - Not recommended

```
type Directions is (North, South, East, West);
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
Heading : Directions;
Today, Tomorrow : Days;
...
Today := Mon;
Today := North; -- compile error
Heading := South;
Heading := East + 1; -- compile error
if Today < Tomorrow then ...
```

# Statements

# Assignment Statements

- Syntax

```
<variable> := <expression>;
```

- Value of expression is copied to target variable
- The type of the RHS must be same as the LHS
  - Rejected at compile-time otherwise

```
type Miles_T is range 0 .. Max_Miles;
type Km_T is range 0 .. Max_Kilometers
...
M : Miles_T := 2; -- universal integer legal for any integer
K : Km_T := 2; -- universal integer legal for any integer
...
M := K; -- compile error
```

# Assignment Statements, Not Expressions

- Separate from expressions
  - No Ada equivalent for these:

```
int a = b = c = 1;
while (line = readline(file))
   { ...do something with line... }
```

- No assignment in conditionals
  - E.g. `if (a == 1)` compared to `if (a = 1)`

# Implicit Range Constraint Checking

- The following code

```
procedure Demo is
  K : Integer;
  P : Integer range 0 .. 100;
begin
  ...
  P := K;
  ...
end Demo;
```

- Generates assignment checks similar to

```
if K < 0 or K > 100 then
  raise Constraint_Error;
else
  P := K;
end if;
```

- Run-time performance impact

# Procedure Calls

- Procedure calls are statements as shown here
- More details in “Subprograms” section

```
procedure Activate (This : in out Foo;
                    Wait : in Boolean);
```

- Traditional call notation

```
Activate (Idle, True);
```

- “Distinguished Receiver” notation
  - For `tagged` types

```
Idle.Activate (True);
```

# Block Statements

```
begin
   Get (V);
   Get (U);
   if U > V then -- swap them
      Swap: declare
         Temp : Integer;
      begin
         Temp := U;
         U := V;
         V := Temp;
      end Swap;
      -- Temp does not exist here
   end if;
   Print (U);
   Print (V);
end;
```

# Conditional Statements

# “If-then-elsif” Statements

- Sequential choice with alternatives
- Avoids `if` nesting
- `elsif` alternatives, tested in textual order
- `else` part still optional

# “If-then-elsif” Example

```
if Valve(N) /= Closed then
  Isolate (Valve(N));
  Failure (Valve (N));
else
  if System = Off then
    Failure (Valve (N));
  end if;
end if;
```

```
if Valve(N) /= Closed then
  Isolate (Valve(N));
  Failure (Valve (N));
elsif System = Off then
  Failure (Valve (N));
end if;
```

# “case” Statements

```
type Directions is  (Forward, Backward, Left, Right);
Direction : Directions;
...
case Direction is
  when Forward =>
    Set_Mode (Drive);
    Go_Forward (1);
  when Backward =>
    Set_Mode (Backup);
    Go_Backward (1);
  when Left =>
    Go_Left (1);
  when Right =>
    Go_Right (1);
end case;
```

***Note**: No fall-through between cases*

# Loop Statements

# Basic Loops

- All kind of loops can be expressed
  - Optional iteration controls
  - Optional exit statements
- Example

```
Wash_Hair : loop
  Lather (Hair);
  Rinse (Hair);
end loop Wash_Hair;
```

# “while-loop” Statements

- Syntax

```
while boolean_expression loop
   sequence_of_statements
end loop;
```

- Identical to

```
loop
   exit when not boolean_expression;
   sequence_of_statements
end loop;
```

- Example

```
while Count < Largest loop
  Count := Count + 2;
  Display (Count);
end loop;
```

# “for-in” Statements

- Successive values of a **discrete** type
  - eg. enumerations values
- Example

```
for Day in Days_T loop
   Refresh_Planning (Day);
end loop;

for Idx in reverse 1 .. 10 loop
   Countdown (Idx);
end loop;
```

# Array Types

# Terminology

- *Index Type*
  - Specifies the values to be used to access the array components
- *Component Type*
  - Specifies the type of values contained by objects of the array type
  - All components are of this same type

```
type Array_T is array (Index_T) of Component_T;
```

# Array Type Index Constraints

- Must be of an integer or enumeration type
- Default to predefined **Integer**
- Allowed to be null range
  - Defines an empty array

```
type Schedule is array (Days range Mon .. Fri) of Float;
type Flags_T is array (-10 .. 10) of Boolean;
type Dynamic is array (1 .. N) of Integer; -- can be null range

subtype Line is String (1 .. 80);
subtype Translation is Matrix (1..3, 1..3);
```

# Run-Time Index Checking

- Array indices are checked at run-time as needed
- Invalid index values result in `Constraint_Error`

```
procedure Test is
   type Int_Arr is array (1..10) of Integer;
   A : Int_Arr;
   K : Integer;
begin
   A := (others => 0);
   K := Foo;
   A (K) := 42; -- runtime error if Foo returns < 1 or > 10
   Put_Line (A(K)'Image);
end Test;
```

# Unconstrained Array Types

# Unconstrained Array Type Declarations

- Do not specify bounds for objects
  - Thus different objects of the same type may have different bounds
- Bounds cannot change once set
- Example

```
type Index is range 1 .. Integer'Last;
type Char_Arr is array (Index range <>)
   of Character;
S1 : Char_Arr(1..10);
S2 : Char_Arr := ('A', 'B', 'C');
```

# “String” Types

- Language-defined unconstrained array types
  - Always have a character component type
  - Always one-dimensional
- Language defines various types
  - **String**, with **Character** as component

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is array (Positive range <>) of Character;
```

  - **Wide_String**, with **Wide_Character** as component
  - **Wide_Wide_String**, with **Wide_Wide_Character** as component
- Can create your own

# Aggregates

# Aggregate “Positional” Form

- Specifies array component values explicitly
- Uses implicit ascending index values

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Working is array (Days) of Boolean;
Week : Working;
...
-- Saturday and Sunday are False, everything else true
Week := (True, True, True, True, True, False, False);
```

# Aggregate "Named" Form

- Explicitly specifies both index and corresponding component values
- Allows any order to be specified
- Ranges and choice lists are allowed (like case choices)

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Working is array (Days) of Boolean;
Week : Working;
...
Week := (Sat => False, Sun => False, Mon..Fri => True);
Week := (Sat | Sun => False, Mon..Fri => True);
```

# “Others”

- Indicates all components not yet assigned a value
- All remaining components get this single value
- Similar to case statement's `others`
- Can be used to apply defaults too

```
type Schedule is array (Days) of Float;
Work : Schedule;
Normal : constant Schedule := (8.0, 8.0, 8.0, 8.0, 8.0,
                               others => 0.0);
```

# Record Types

# Syntax and Examples

- Syntax (simplified)

```
type T is record
   Component_Name : Type [:= Default_Value];
   ...
end record;

type T_Empty is null record;
```

- Example

```
type Record1_T is record
   Field1 : integer;
   Field2 : boolean;
end record;
```

- Records can be **discriminated** as well

```
type T (Size : Natural := 0) is record
   Text : String (1 .. Size);
end record;
```

# Dot Notation for Component Reference

```
type Months_T is (January, February, ..., December);
type Date is record
   Day   : Integer range 1 .. 31;
   Month : Months_T;
   Year  : Integer range 0 .. 2099;
end record;
Arrival : Date;
...
Arrival.Day := 27;  -- components referenced by name
Arrival.Month := November;
Arrival.Year := 1990;
```

- Can reference nested components

```
Employee
   .Birth_Date
     .Month := March;
```

# Aggregates

# Aggregates

- Literal values for composite types
  - As for arrays
  - Default value / selector: `<>`, `others`
- Can use both **named** and **positional**
  - Unambiguous
- Example:

```
(Pos_1_Value,
 Pos_2_Value,
 Component_3 => Pos_3_Value,
 Component_4 => <>,
 others => Remaining_Value)
```

# Record Aggregate Examples

```
type Color_T is (Red);
type Car_T is record
   Color    : Color_T;
   Plate_No : String (1 .. 6);
   Year     : Natural;
end record;
type Complex_T is record
   Real      : Float;
   Imaginary : Float;
end record;
declare
   Car   : Car_T     := (Red, "ABC123", Year => 2_022);
   Phase : Complex_T := (1.2, 3.4);
begin
   Phase := (Real => 5.6, Imaginary => 7.8);
end;
```

# Default Values

# Component Default Values

```
type Complex is
  record
    Real : Float := 0.0;
    Imaginary : Float := 0.0;
  end record;
-- all components use defaults
Phasor : Complex;
-- all components must be specified
I : constant Complex := (0.0, 1.0);
```

# Defaults Within Record Aggregates

- Specified via the *box* notation
- Value for the component is thus taken as for a stand-alone object declaration
  - So there may or may not be a defined default!
- Can only be used with “named association” form
  - But can mix forms, unlike array aggregates

```
type Complex is
   record
      Real : Float := 0.0;
      Imaginary : Float := 0.0;
   end record;
Phase := (42.0, Imaginary => <>);
```

# Subprograms

# Introduction

- Are syntactically distinguished as `function` and `procedure`
  - Functions represent *values*
  - Procedures represent *actions*

```
function Is_Leaf (T : Tree) return Boolean
procedure Split (T : in out Tree;
                 Left : out Tree;
                 Right : out Tree)
```

- Provide direct syntactic support for separation of specification from implementation

```
function Is_Leaf (T : Tree) return Boolean;

function Is_Leaf (T : Tree) return Boolean is
begin
...
end Is_Leaf;
```

# Parameters

# Parameter Associations In Calls

- Associate formal parameters with actuals
- Both positional and named association allowed

```
Something (ActualX, Formal2 => ActualY);
Something (Formal2 => ActualY,
           Formal1 => ActualX);
```

- Having named followed by positional is forbidden

```
-- Compilation Error
Something (Formal1 => ActualX, ActualY);
```

# Parameter Modes and Return

- Mode `in`
  - Actual parameter is `constant`
  - Can have **default**, used when **no value** is provided

```
procedure P (N : in Integer := 1; M : in Positive);
...
P (M => 2);
```

- Mode `out`
  - Writing is **expected**
  - Reading is **allowed**
  - Actual **must** be a writable object
- Mode `in out`
  - Actual is expected to be **both** read and written
  - Actual **must** be a writable object
- Function `return`
  - **Must** always be handled

# Nested Subprograms

# Nested Subprogram Example

```
procedure Main is

   function Read (Prompt : String) return Types.Line_T is
   begin
      Put ("> ");
      return Types.Line_T'Value (Get_Line);
   end Read;

   Lines : Types.Lines_T (1 .. 10);
begin
   for J in Lines'Range loop
      Lines (J) := Read ("Line " & J'Image);
   end loop;
end Main;
```

# Packages

# Declarations

# Package Declarations

- Required in all cases
  - Cannot have a package without the declaration
- Describe the client's interface
  - Declarations are exported to clients
  - Effectively the "pin-outs" for the black-box
- When changed, requires client's recompilation
  - The "pin-outs" have changed

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;

package Data is
   Object : integer;
end Data;
```

# Compile-Time Visibility Control

- Items in the declaration are visible to users

```
package Some_Package is
   -- exported declarations of
   --   types, variables, subprograms ...
end Some_Package;
```

- Items in the body are never externally visible
  - Compiler prevents external references

```
package body Some_Package is
   -- hidden declarations of
   --   types, variables, subprograms ...
   -- implementations of exported subprograms etc.
end Some_Package;
```

# Example of Exporting To Clients

- Variables, types, exception, subprograms, etc.
  - The primary reason for separate subprogram declarations

```
package P is
   procedure This_Is_Exported;
end P;

package body P is
   procedure Not_Exported is
      ...
   procedure This_Is_Exported is
      ...
end P;
```

# Referencing Exported Items

- Achieved via “dot notation”
- Package Specification

```
package Float_Stack is
  Max : constant := 100;
  procedure Push (X : in Float);
  procedure Pop (X : out Float);
end Float_Stack;
```

- Package Reference

```
with Float_Stack;
procedure Test is
   X : Float;
begin
   Float_Stack.Pop (X);
   Float_Stack.Push (12.0);
   if Count < Float_Stack.Max then ...
```

# “use” Clauses

- Provide direct visibility into packages' exported items
  - *Direct Visibility* - as if object was referenced from within package being used
- May still use expanded name

```
package Ada.Text_IO is
  procedure Put_Line(...);
  procedure New_Line(...);
  ...
end Ada.Text_IO;

with Ada.Text_IO;
procedure Hello is
  use Ada.Text_IO;
begin
  Put_Line("Hello World");
  New_Line(3);
  Ada.Text_IO.Put_Line ("Good bye");
end Hello;
```

# Private Types

# Implementing Abstract Data Types via Views

# Declaring Private Types for Views

- Partial syntax

```
type defining_identifier is private;
```

- Private type declaration must occur in visible part
  - *Partial View*
  - Only partial information on the type
  - Users can reference the type name
- Full type declaration must appear in private part
  - Completion is the *Full View*
  - **Never** visible to users
  - **Not** visible to designer until reached

```
package Bounded_Stacks is
   type Stack is private;
   procedure Push (Item : in Integer; Onto : in out Stack);
   ...
private
   ...
   type Stack is record
      Top : Positive;
      ...
 end Bounded_Stacks;
```

# Users Declare Objects of the Type

```
X, Y, Z : Stack;
...
Push (42, X);
...
if Empty (Y) then
...
Pop (Counter, Z);
```

# Compile-Time Visibility Protection

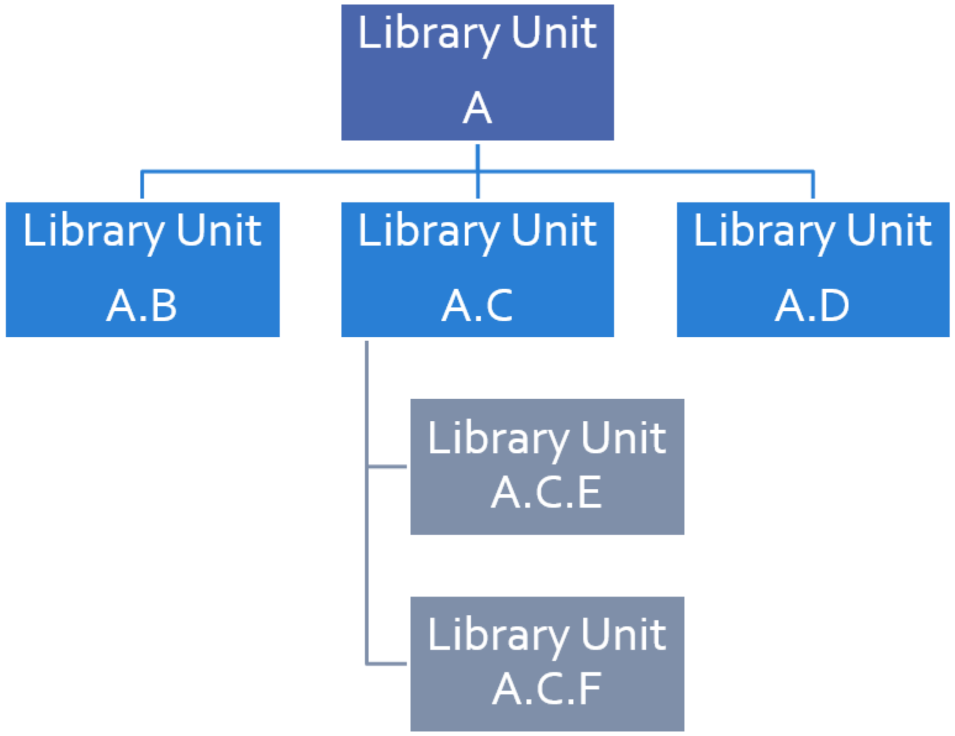
- No type representation details available outside the package
- Therefore users cannot compile code referencing representation
- This does not compile

```
with Bounded_Stacks;
procedure User is
   S : Bounded_Stacks.Stack;
begin
   S.Top := 1;  -- Top is not visible
end User;
```

# Program Structure

# Hierarchical Library Units

- Address extensibility issue
  - Can extend packages with visibility to parent private part
  - Extensions do not require recompilation of parent unit
  - Visibility of parent's private part is protected
- Directly support subsystems
  - Extensions all have the same ancestor *root* name



# Programming By Extension

- Parent unit

```
package Complex is
  type Number is private;
  function "*" (Left, Right : Number) return Number;
  function "/" (Left, Right : Number) return Number;
  function "+" (Left, Right : Number) return Number;
  function "-" (Left, Right : Number) return Number;
...
private
  type Number is record
    Real_Part, Imaginary_Part : Float;
  end record;
end Complex;
```

- Extension created to work with parent unit

```
package Complex.Utils is
  procedure Put (C : in Number);
  function As_String (C : Number) return String;
  ...
end Complex.Utils;
```

# Extension Can See Private Section

- With certain limitations

```
with Ada.Text_IO;
package body Complex.Utils is
  procedure Put(C : in Number) is
  begin
    Ada.Text_IO.Put(As_String(C));
  end Put;
  function As_String(C : Number) return String is
  begin
    -- Real_Part and Imaginary_Part are
    -- visible to child's body
    return "(" & Float'Image(C.Real_Part) & ", " &
           Float'Image(C.Imaginary_Part) & ")";
  end As_String;
...
end Complex.Utils;
```

# Predefined Hierarchies

- Standard library facilities are children of **Ada**
  - **Ada.Text_IO**
  - **Ada.Calendar**
  - **Ada.Command_Line**
  - **Ada.Exceptions**
  - et cetera
- Other root packages are also predefined
  - **Interfaces.C**
  - **Interfaces.Fortran**
  - **System.Storage_Pools**
  - **System.Storage_Elements**
  - et cetera

# Genericity

# The Notion of a Pattern

- Sometimes algorithms can be abstracted from types and subprograms

```
procedure Swap_Int (Left, Right : in out Integer) is
  V : Integer := Left;
begin
   Left := Right;
   Right := V;
end Swap_Int;

procedure Swap_Bool (Left, Right : in out Boolean) is
   V : Boolean := Left;
begin
   Left := Right;
   Right := V;
end Swap_Bool;
```

- It would be nice to extract these properties in some common pattern, and then just replace the parts that need to be replaced

```
procedure Swap (Left, Right : in out (Integer | Boolean)) is
  V : (Integer | Boolean) := Left;
begin
   Left := Right;
   Right := V;
end Swap;
```

# Ada Generic Compared to C++ Template

## Ada Generic

```
-- specification
generic
   type T is private;
procedure Swap (L, R : in out T);

-- implementation
procedure Swap (L, R : in out T) is
   Tmp : T := L;
begin
   L := R;
   R := Tmp;
end Swap;

-- instance
procedure Swap_F is new Swap (Float);
```

## C++ Template

```
// prototype
template <class T>
void Swap (T & L, T & R);

// implementation
template <class T>
void Swap (T & L, T & R) {
   T Tmp = L;
   L = R;
   R = Tmp;
}

// instance
int x, y;
Swap<int>(x,y);
```

*Works for Ada packages as well (similar to templates for C++ classes)*

# Generic Contracts

# Definitions

- A formal generic parameter is a template
- Properties are either *Constraints* or *Capabilities*
  - Expressed from the **body** point of view
  - Constraints: e.g. unconstrained, `limited`
  - Capabilities: e.g. `tagged`, primitives

```
generic
   type Pv is private;           -- allocation, copy, assignment, "="
   with procedure Sort (T : Pv); -- primitive of Pv
   type Unc (<>) is private;     -- allocation require a value
   type Lim is limited private;  -- no copy or comparison
   type Disc is (<>);            -- 'First, ordering
package Generic_Pkg is [...]
```

- Actual parameter **may** require constraints, and **must** provide capabilities

```
package Pkg is new Generic_Pkg (
   Pv => Integer, -- has capabilities of private
   Sort => Sort -- procedure Sort (T : Integer)
   Unc => String,  -- uses "unconstrained" constraint
   Lim => Float,   -- does not use "limited" constraint
   Disc => Boolean, -- has capability of discrete
);
```

# Generic Formal Data

# Constants and Variables Parameters

- Variables can be specified on the generic contract
- The mode specifies the way the variable can be used:
  - `in` → read only
  - `in out` → read write
- Generic variables can be defined after generic types

```
generic
    type T is private;
    X1 : Integer;   -- constant
    X2 : in out T;  -- variable
procedure P;

V : Float;

procedure P_I is new P
    (T  => Float,
     X1 => 42,
     X2 => V);
```

# Generic Subprogram Parameters

- Subprograms can be defined in the generic contract
- Must be introduced by `with` to differ from the generic unit

```
generic
   with procedure Callback;
procedure P;
procedure P is
begin
   Callback;
end P;
procedure Something;
procedure P_I is new P (Something);
```

# Ada Contracts

# Design-By-Contract

- Source code acting in roles of **client** and **supplier** under a binding **contract**
  - *Contract* specifies *requirements* or *guarantees*

    *"A specification of a software element that affects its use by potential clients."* (Bertrand Meyer)
  - *Supplier* provides services
    - Guarantees specific functional behavior
    - Has requirements for guarantees to hold
  - *Client* utilizes services
    - Guarantees supplier's conditions are met
    - Requires result to follow the subprogram's guarantees

# Assertion

- Boolean expression expected to be `True`
- Said *to hold* when `True`

```
pragma Assert (not Full (Stack));
-- stack is not full
pragma Assert (Stack_Length = 0,
               Message => "stack was not empty");
-- stack is empty
```

- Raises language-defined `Assertion_Error` exception if expression does not hold

# Defensive Programming

- Should be replaced by subprogram contracts when possible

```
procedure Push (S : Stack) is
   Entry_Length : constant Positive := Length (S);
begin
   pragma Assert (not Is_Full (S)); -- entry condition
   [...]
   pragma Assert (Length (S) = Entry_Length + 1); -- exit condition
end Push;
```

- Subprogram contracts are an **assertion** mechanism
  - **Not** a drop-in replacement for all defensive code

```
procedure Force_Acquire (P : Peripheral) is
begin
   if not Available (P) then
      -- Corrective action
      Force_Release (P);
      pragma Assert (Available (P));
   end if;
   Acquire (P);
end;
```

# Preconditions and Postconditions

# Subprogram-based Assertions

- **Explicit** part of a subprogram's **specification**
  - Unlike defensive code
- *Precondition*
  - Assertion expected to hold **prior to** subprogram call
- *Postcondition*
  - Assertion expected to hold **after** subprogram return
- Requirements and guarantees on both supplier and client
- Syntax uses **aspects**

```
procedure Push (This : in out Stack_T;
                Value : Content_T)
  with Pre  => not Full (This),
       Post => not Empty (This)
               and Top (This) = Value;
```

# Postcondition 'Old Attribute

- Values as they were just before the call
- Uses language-defined attribute `'Old`
  - Can be applied to most any visible object
    - `limited` types are forbidden
    - May be expensive
  - Expression can be **arbitrary**
    - Typically `out`, `in out` parameters and globals

```
procedure Increment (This : in out Integer) with
    Pre  => This < Integer'Last,
    Post => This = This'Old + 1;
```

# Function Postcondition 'Result Attribute

- function result can be manipulated with 'Result

```
function Greatest_Common_Denominator
  (A, B : Integer)
  return Integer with
    Pre  =>  A > 0 and B > 0,
    Post =>  Is_GCD
               (A, B,
                Greatest_Common_Denominator'Result);
```

# Type Invariants

# Strong Typing

- Ada supports strong typing

```
type Small_Integer_T is range -1_000 .. 1_000;
type Enumerated_T is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
type Array_T is array (1 .. 3) of Boolean;
```

- What if we need stronger enforcement?
  - Number must be even
  - Subset of non-consecutive enumerals
  - Array should always be sorted
- **Type Invariant**
  - Property of type that is always true on **external** reference
  - *Guarantee* to client, similar to subprogram postcondition
- **Subtype Predicate**
  - Property of type that is always true, unconditionally
  - Can add arbitrary constraints to a type, unlike the “basic” type system

# Example Type Invariant

- A bank account balance must always be consistent
  - Consistent Balance: Total Deposits - Total Withdrawals = Balance

```
package Bank is
   type Account is private with
     Type_Invariant => Consistent_Balance (Account);
   ...
   -- Called automatically for all Account objects
   function Consistent_Balance (This : Account)
     return Boolean;
   ...
private
   ...
end Bank;
```

# Invariants Don't Apply Internally

- No checking within supplier package
  - Otherwise there would be no way to implement anything!
- Only matters when clients can observe state

```
procedure Open (This : in out Account;
                Name : in String;
                Initial_Deposit : in Currency) is
begin
   This.Owner := To_Unbounded_String (Name);
   This.Current_Balance := Initial_Deposit;
   -- invariant would be false here!
   This.Withdrawals := Transactions.Empty_List;
   This.Deposits := Transactions.Empty_List;
   This.Deposits.Append (Initial_Deposit);
   -- invariant is now true
end Open;
```

# Subtype Predicates

# Predicates

- Assertion expected to hold for all objects of given type
- Expressed as any legal boolean expression in Ada
  - Quantified and conditional expressions
  - Boolean function calls
- Two forms in Ada
  - **Static Predicates**
    - Specified via aspect named `Static_Predicate`
  - **Dynamic Predicates**
    - Specified via aspect named `Dynamic_Predicate`
- Can apply to `type` or `subtype`

# Subtype Predicate Examples

- Dynamic Predicate

```
subtype Even is Integer with Dynamic_Predicate =>
   Even mod 2 = 0; -- Boolean expression
   -- (Even indicates "current instance")
```

- Static Predicate

```
type Serial_Baud_Rate is range 110 .. 115200
  with Static_Predicate => Serial_Baud_Rate  in
    -- Non-contiguous range
    110   | 300   | 600 | 1200 | 2400 | 4800 |
    9600 | 14400 | 19200 | 28800 | 38400 | 56000 |
    57600 | 115200;
```

# Predicate Checking

- Calls inserted automatically by compiler
- Violations raise exception `Assertion_Error`
  - When predicate does not hold (evaluates to False)
- Checks are done before value change
  - Same as language-defined constraint checks
- Associated variable is unchanged when violation is detected

# Tasking

# A Simple Task

- Concurrent code execution via `task`

```
procedure Main is
   task type Put_T;
   task body Put_T is
   begin
      loop
         delay 1.0;
         Put_Line ("T");
      end loop;
   end Put_T;

   T : Put_T;
begin -- Main task body
   loop
      delay 1.0;
      Put_Line ("Main");
   end loop;
end;
```

- A task is started when its declaration scope is **elaborated**
- Its enclosing scope exits when **all tasks** have finished

# Two Synchronization Models

- Active
  - Rendezvous
  - **Client / Server** model
  - Server **entries**
  - Client **entry calls**
- Passive
  - **Protected objects** model
  - Concurrency-safe **semantics**

# Tasks

# Rendezvous Definitions

- **Server** declares several `entry`
- Client calls entries like subprograms
- Server `accept` the client calls
- At each standalone `accept`, server task **blocks**
  - **Until** a client calls the related `entry`

```
task type Msg_Box_T is
   entry Start;
   entry Receive_Message (S : String);
end Msg_Box_T;

task body Msg_Box_T is
begin
   loop
      accept Start;
      Put_Line ("start");

      accept Receive_Message (S : String) do
         Put_Line (S);
      end Receive_Message;
   end loop;
end Msg_Box_T;
```

# Rendezvous Entry Calls

- Upon calling an `entry`, client **blocks**
  - **Until** server reaches `end` of its `accept` block

```
T : Msg_Box_T;

Put_Line ("calling start");
T.Start;
Put_Line ("calling receive 1");
T.Receive_Message ("1");
Put_Line ("calling receive 2");
T.Receive_Message ("2");
```

- May be executed as follows:

```
calling start
start               -- May switch place with line below
calling receive 1   -- May switch place with line above
Receive 1
calling receive 2
-- Blocked until another task calls Start
```

# Accepting a Rendezvous

- `accept` statement
  - Wait on single entry
  - If entry call waiting: Server handles it
  - Else: Server **waits** for an entry call
- `select` statement
  - **Several** entries accepted at the **same time**
  - Can **time-out** on the wait
  - Can be **not blocking** if no entry call waiting
  - Can **terminate** if no clients can **possibly** make entry call
  - Can **conditionally** accept a rendezvous based on a **guard expression**

# Protected Objects

# Protected Objects

- **Multitask-safe** accessors to get and set state
- **No** direct state manipulation
- **No** concurrent modifications

# Protected Objects Example

```
protected type Some_Value is
   procedure Set
      (V : Integer);
   function Get
      return Integer;
private
   Value : Integer;
end Some_Value;
```

```
protected body Some_Value is
   procedure Set
      (V : Integer) is
   begin
      Value := V;
   end Set;

   function Get
      return Integer is
   begin
      return Value;
   end Get;
end Some_Value;
```

# Protected: Functions and Procedures

- A `function` can **get** the state
  - Protected data is **read-only**
  - Concurrent call to `function` is **allowed**
  - **No** concurrent call to `procedure`
- A `procedure` can **set** the state
  - **No** concurrent call to either `procedure` or `function`
  - In case of concurrency, other callers get **blocked**
    - Until call finishes

# Delays

# Delay keyword

- `delay` keyword part of tasking
- Blocks for a time
- Relative: Blocks for at least `Duration`
- Absolute: Blocks until a given `Calendar.Time` or `Real_Time.Time`

```
procedure Main is
   Relative : Duration := 1.0;
   Absolute : Calendar.Time
     := Calendar.Time_Of (2030, 10, 01);
begin
   delay Relative;
   delay until Absolute;
end Main;
```

# Access Types

# Access Types Design

- Memory-addressed objects are called *Access Types*
- C++

```
int *P_C =
      malloc (sizeof (int));
int *P_CPP = new int;
int *G_C = &Some_Int;
```

- Ada

```
type Integer_General_Access
  is access all Integer;
G : aliased Integer;
G_A : Integer_General_Access := G'access;
```

# Null Values

- A pointer that does not point to any actual data has a `null` value
- Access types have default value of `null`
- `null` can be used in assignments and comparisons

```
declare
   type Acc is access all Integer;
   V : Acc;
begin
   if V = null then
      --  will go here
   end if
   V := new Integer'(0);
   V := null; -- semantically correct, but memory leak
```

# Dereferencing Access Types

- `.all` does the access dereference
  - Lets you access the object pointed to by the pointer
- `.all` is optional for
  - Access on a component of an array
  - Access on a component of a record

```
type Record_T is record
  F1, F2 : Integer;
end record;
type Integer_Access_T is access Integer;
type String_Access_T is access all String;
type Record_Access_T is access all Record_T;

Integer_Access : Integer_Access_T := new Integer;
String_Access  : String_Access_T  := new String'("abc");
Record_Access  : Record_Access_T  := new R;

Integer_Access.all := 123;
String_Access(1)   := "-";
Record_Access.F1   := 456;
Record_Access.all  := (7, 8);
```

# General Access Types

# General Access Types

- Can point to any pool (including stack)

```
type T is [...]
type T_Access is access all T;
V : T_Access := new T;
```

- Still distinct type
- Conversions are possible

```
type T_Access_2 is access all T;
V2 : T_Access_2 := T_Access_2 (V); -- legal
```

# Allocations

- Objects are created with the `new` reserved word
- The created object must be constrained
  - The constraint is given during the allocation

```
V : String_Access := new String (1 .. 10);
```

- The object can be created by copying an existing object - using a qualifier

```
V : String_Access :=
   new String'("This is a String");
```

# Deallocations

- Deallocations are unsafe
  - Multiple deallocations problems
  - Memory corruptions
  - Access to deallocated objects
- As soon as you use them, you lose the safety of your access
- But sometimes, you have to do what you have to do ...
  - There's no simple way of doing it
  - Ada provides **Ada.Unchecked_Deallocation**
  - Has to be instantiated (it's a generic)
  - Must work on an object, reset to `null` afterwards

# Referencing The Stack

- By default, stack-allocated objects cannot be referenced - and can even be optimized into a register by the compiler
- `aliased` declares an object to be referenceable through an access value

```
V : aliased Integer;
```

- `'Access` attribute gives a reference to the object

```
A : Int_Access := V'Access;
```

  - `'Unchecked_Access` does it **without checks**

# Tagged Types

# Derivation Ada vs C++

```ada
type T1 is tagged record
  Member1 : Integer;
end record;

procedure Attr_F (This : T1);

type T2 is new T1
  with record
    Member2 : Integer;
  end record;

overriding
procedure Attr_F
    (This : T2);
procedure Attr_F2
    (This : T2);
```

```cpp
class T1 {
  public:
    int Member1;
    virtual void Attr_F(void);
  };

class T2 : public T1 {
  public:
    int Member2;
    virtual void Attr_F(void);
    virtual void Attr_F2(void);
  };
```

# Tagged Derivation

# Difference with Simple Derivation

- Tagged derivation **can** change the structure of a type
  - Keywords `tagged record` and `with record`

```
type Root is tagged record
   F1 : Integer;
end record;

type Child is new Root with record
   F2 : Integer;
end record;
```

# Type Extension

- A tagged derivation **has** to be a type extension
  - Use `with null record` if there are no additional components

```
type Child is new Root with null record;
type Child is new Root; -- illegal
```

- Conversion is only allowed from **child to parent**

```
V1 : Root;
V2 : Child;
...
V1 := Root (V2);
V2 := Child (V1); -- illegal
```

# Prefix Notation

- Tagged types primitives can be called as usual
- The call can use prefixed notation
  - **If** the first argument is a controlling parameter
  - No need for `use` or `use type` for visibility

```
-- Prim1 visible even without *use Pkg*
X.Prim1;

declare
   use Pkg;
begin
   Prim1 (X);
end;
```

# Interfacing with C

# Import / Export

# Import / Export Aspects (1/2)

- Aspect `Import` allows a C implementation to complete an Ada specification
  - Ada view

```
procedure C_Proc
   with Import,
        Convention    => C,
        External_Name => "c_proc";
```

  - C implementation

```
void SomeProcedure (void) {
   // some code
}
```

- Aspect `Export` allows an Ada implementation to complete a C specification
  - Ada implementation

```
procedure Some_Procedure
   with Export,
        Convention    => C,
        External_Name => "ada_some_procedure");

procedure Some_Procedure is
begin
  -- some code
end Some_Procedure;
```

  - C view

```
extern void ada_some_procedure (void);
```

# Import / Export Aspects (2/2)

- You can also import/export variables
  - Variables imported won't be initialized
  - Ada view

```
My_Var : Integer_Type
   with Import,
        Convention    => C,
        External_Name => "my_var");
```

  - C implementation

```
int my_var;
```

# Parameter Passing

# Passing Scalar Data as Parameters

- C types are defined by the standard
- Ada types are implementation-defined
- GNAT standard types are compatible with C types
  - Implementation choice, use carefully
- At the interface level, scalar types must be either constrained with representation clauses, or coming from Interfaces.C
- Ada view

```ada
with Interfaces.C;
function C_Proc (I : Interfaces.C.Int)
    return Interfaces.C.Int
    with Import,
         Convention    => C,
         External_Name => "c_proc");
```

- C view

```c
int c_proc (int i) {
  /* some code */
}
```

# Passing Structures as Parameters

- An Ada record that is mapping on a C struct must:
  - Be marked as convention C to enforce a C-like memory layout
  - Contain only C-compatible types
- C View

```c
enum Enum {E1, E2, E3};
struct Rec {
   int A, B;
   Enum C;
};
```

- Ada View

```ada
type Enum is (E1, E2, E3) with Convention => C;
type Rec is record
  A, B : int;
  C : Enum;
end record with Convention => C;
```

# Parameter modes

- `in` scalar parameters passed by copy
- `out` and `in out` scalars passed using temporary pointer on C side
- By default, composite types passed by reference on all modes except when the type is marked `C_Pass_By_Copy`
  - Be very careful with records - some C ABI pass small structures by copy!

# Polymorphism

# Classes

- In Ada, a Class denotes an inheritance subtree
- Class of **T** is the class of **T** and all its children
- Type `T'Class` can designate any object typed after type of class of **T**

```ada
type Root is tagged null record;
type Child1 is new Root with null record;
type Child2 is new Root with null record;
type Grand_Child1 is new Child1 with null record;
-- Root'Class = {Root, Child1, Child2, Grand_Child1}
-- Child1'Class = {Child1, Grand_Child1}
-- Child2'Class = {Child2}
-- Grand_Child1'Class = {Grand_Child1}
```

- Objects of type `T'Class` have at least the properties of T
  - Fields of **T**
  - Primitives of **T**

# Abstract Types

- A tagged type can be declared `abstract`
- Then, `abstract tagged` types:
  - cannot be instantiated
  - can have abstract subprograms (with no implementation)
  - Non-abstract derivation of an abstract type must override and implement abstract subprograms

# 'Class and Prefix Notation

Prefix notation rules apply when the first parameter is of a class wide type

```
type Root is tagged null record;
procedure P (V : Root'Class);
type Child is new Root with null record;

V1 : Root;
V2 : Root'Class := Root'(others => <>);
...
P (V1);
P (V2);
V1.P;
V2.P;
```

# Dispatching and Redispatching

# Calls on class-wide types (1/2)

- Any subprogram expecting a T object can be called with a `T'Class` object

```
type Root is tagged null record;
procedure P (V : Root);

type Child is new Root with null record;
procedure P (V : Child);

   V1 : Root'Class := [...]
   V2 : Child'Class := [...]
begin
   P (V1);
   P (V2);
```

# Calls on class-wide types (2/2)

- **The *actual* type of the object is not known at compile time**
- **The *right* type will be selected at runtime**

*Ada*

```
declare
   V1 : Root'Class :=
        Root'(others => <>);
   V2 : Root'Class :=
        Child'(others => <>);
begin
   V1.P; -- calls P of Root
   V2.P; -- calls P of Child
```

*C++*

```
Root * V1 = new Root ();
Root * V2 = new Child ();
V1->P ();
V2->P ();
```

# Low Level Programming

https://public-training.adacore.com/2024-10-21.html

adacore.com

# Data Representation

# Data Representation vs Requirements

- Developer usually defines requirements on a type

```
type My_Int is range 1 .. 10;
```

- The compiler then generates a representation for this type that can accommodate requirements
  - In GNAT, can be consulted using `-gnatR2` switch

```
type Ada2012_Int is range 1 .. 10
   with Object_Size => 8,
        Value_Size  => 4,
        Alignment   => 1;
```

- These values can be explicitly set, the compiler will check their consistency
- They can be queried as attributes if needed

```
X : Integer := My_Int'Alignment;
```

# Value_Size / Size

- **Value_Size** (or **Size** in the Ada Reference Manual) is the minimal number of bits required to represent data
  - For example, `Boolean'Size = 1`
- The compiler is allowed to use larger size to represent an actual object, but will check that the minimal size is enough

```
type T1 is range 1 .. 4
   with Size => 3;
```

# Alignment

- Number of bytes on which the type has to be aligned
- Some alignment may be more efficient than others in terms of speed (e.g. boundaries of words (4, 8))
- Some alignment may be more efficient than others in terms of memory usage

```
type T1 is range 1 .. 4
   with Size      => 4,
        Alignment => 8;
```

# Pack Aspect

- **pack** aspect applies to composite types (record and array)
- Compiler optimizes data for size no matter performance impact
- Unpacked

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Enum;
end record;
type Ar is array (1 .. 1000) of Boolean;
-- Rec'Size is 48, Ar'Size is 8000
```

- Packed

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Enum;
end record with Pack;
type Ar is array (1 .. 1000) of Boolean;
pragma Pack (Ar);
-- Rec'Size is 35, Ar'Size is 1000
```

# Record Representation Clauses

- Exact mapping between a record and its binary representation
- Optimization purposes, or hardware requirements
  - Driver mapped on the address space, communication protocol...
- Fields represented as

```
<name> at <byte> range
   <starting-bit> ..
   <ending-bit>
```

```
type Rec1 is record
   A : Integer range 0 .. 4;
   B : Boolean;
   C : Integer;
   D : Enum;
end record;
for Rec1 use record
   A at 0 range 0 ..  2;
   B at 0 range 3 ..  3;
   C at 0 range 4 .. 35;
   -- unused space here
   D at 5 range 0 ..  2;
end record;
```

# Array Representation Clauses

- `Component_Size` for array's **component's** size

```
type Ar2 is array (1 .. 1000) of Boolean
    with Component_Size => 2;
```

# Address Clauses and Overlays

# Address Clauses

- Ada allows specifying the address of an entity

```
Var : Unsigned_32 with Address => 16#1234_ABCD#;
```

- Very useful to declare I/O registers
  - For that purpose, the object should be declared volatile:

```
Var : Unsigned_32 with Volatile;
```

- Useful to read a value anywhere

```
function Get_Byte (Addr : Address) return Unsigned_8 is
  V : Unsigned_8 with Volatile, Address => Addr;
begin
  return V;
end;
```

  - In particular the address doesn't need to be constant
  - But must match alignment

# Unchecked Conversion

- **Unchecked_Conversion** allows an unchecked *bitwise* conversion of data between two types
- Needs to be explicitly instantiated

```
type Bitfield is array (1 .. Integer'Size) of Boolean;
function To_Bitfield is new
   Ada.Unchecked_Conversion (Integer, Bitfield);
V  : Integer;
V2 : Bitfield := To_Bitfield (V);
```

- Avoid conversion if the sizes don't match
  - Not defined by the standard
  - Many compilers will warn if the type sizes do not match

# Inline Assembly

# Simple Statement

- Instruction without inputs/outputs

```
Asm ("halt", Volatile => True);
```

  - You may specify **Volatile** to avoid compiler optimizations
  - In general, keep it False unless it created issues
- You can group several instructions

```
Asm ("nop" & ASCII.LF & ASCII.HT
     & "nop", Volatile => True);
Asm ("nop; nop", Volatile => True);
```

- The compiler doesn't check the assembly, only the assembler will
  - Error message might be difficult to read

# Instruction Counter Example (x86)

```
with System.Machine_Code; use System.Machine_Code;
with Ada.Text_IO;         use Ada.Text_IO;
with Interfaces;          use Interfaces;
procedure Main is
   Low   : Unsigned_32;
   High  : Unsigned_32;
   Value : Unsigned_64;
   use ASCII;
begin
   Asm ("rdtsc" & LF,
        Outputs =>
           (Unsigned_32'Asm_Output ("=g", Low),
            Unsigned_32'Asm_Output ("=a", High)),
        Volatile => True);
   Values := Unsigned_64 (Low) +
             Unsigned_64 (High) * 2 ** 32;
   Put_Line (Values'Image);
end Main;
```