Ada Basic Types - Advanced

Subtypes - Full Picture

Implicit Subtype

- The declaration

```
type Typ is range L .. R;
```

- Is short-hand for

```
type <Anon> is new Predefined_Integer_Type;
subtype Typ is <Anon> range L .. R;
```

- `<Anon>` is the *Base* type of `Typ`
  - Accessed with `Typ'Base`

Implicit Subtype Explanation

```
type <Anon> is new Predefined_Integer_Type;
subtype Typ is <Anon> range L .. R;
```

- Compiler choses a standard integer type that includes `L .. R`
  - `Integer`, `Short_Integer`, `Long_Integer`, etc.
  - **Implementation-defined** choice, non portable
- New anonymous type `<Anon>` is derived from the predefined type
- `<Anon>` inherits the type's operations (+, - ...)
- `Typ`, subtype of `<Anon>` is created with `range L .. R`
- `Typ'Base` will return the type `<Anon>`

Stand-Alone (Sub)Type Names

- Denote all the values of the type or subtype
    - Unless explicitly constrained

```
subtype Constrained_Sub is Integer range 0 .. 10;
subtype Just_A_Rename is Integer;
X : Just_A_Rename;
...
for I in Constrained_Sub loop
  X := I;
end loop;
```

Subtypes Localize Dependencies

- Single points of change
- Relationships captured in code
- No subtypes

```
type List is array (1 .. 12) of Some_Type;

K : Integer range 0 .. 12 := 0; -- anonymous subtype
Values : List;
...
if K in 1 .. 12 then ...
for J in Integer range 1 .. 12 loop ...
```

- Subtypes

```
type Counter is range 0 .. 12;
subtype Index is Counter range 1 .. Counter'Last;
type List is array (Index) of Some_Type;

K : Counter := 0;
Values : List;
...
if K in Index then ...
for J in Index loop ...
```

Subtypes May Enhance Performance

- Provides compiler with more information
- Redundant checks can more easily be identified

```
subtype Index is Integer range 1 .. Max;
type List is array (Index) of Float;
K : Index;
Values : List;
...
K := Some_Value;    -- range checked here
Values (K) := 0.0; -- so no range check needed here
```

Subtypes Don't Cause Overloading

- Illegal code: re-declaration of **F**

```
type A is new Integer;
subtype B is A;
function F return A is (0);
function F return B is (1);
```

Subtypes and Default Initialization

Ada 2012

- Not allowed: Defaults on new `type` only
  - `subtype` is still the same type
- **Note:** Default value may violate subtype constraints
  - Compiler error for static definition
  - `Constraint_Error` otherwise

```
type Tertiary_Switch is (Off, On, Neither)
   with Default_Value => Neither;
subtype Toggle_Switch is Tertiary_Switch
    range Off .. On;
Safe : Toggle_Switch := Off;
Implicit : Toggle_Switch; -- compile error: out of range
```

Attributes Reflect the Underlying Type

```
type Color is
    (White, Red, Yellow, Green, Blue, Brown, Black);
subtype Rainbow is Color range Red .. Blue;
```

- `T'First` and `T'Last` respect constraints
  - `Rainbow'First` → Red *but* `Color'First` → White
  - `Rainbow'Last` → Blue *but* `Color'Last` → Black
- Other attributes reflect base type
  - `Color'Succ (Blue)` = Brown = `Rainbow'Succ (Blue)`
  - `Color'Pos (Blue)` = 4 = `Rainbow'Pos (Blue)`
  - `Color'Val (0)` = White = `Rainbow'Val (0)`
- Assignment must still satisfy target constraints

```
Shade : Color range Red .. Blue := Brown; -- runtime error
Hue : Rainbow := Rainbow'Succ (Blue);     -- runtime error
```

Quiz

```
1 type T1 is range 0 .. 10;
2 function "-" (V : T1) return T1;
3 subtype T2 is T1 range 1 .. 9;
4 function "-" (V : T2) return T2;
5
6 Obj : T2 := -T2 (1);
```

Which function is executed at line 6?

A. The one at line 2
B. The one at line 4
C. A predefined `"-"` operator for integer types
D. None: The code is illegal

Quiz

```
1 type T1 is range 0 .. 10;
2 function "-" (V : T1) return T1;
3 subtype T2 is T1 range 1 .. 9;
4 function "-" (V : T2) return T2;
5
6 Obj : T2 := -T2 (1);
```

Which function is executed at line 6?

A. The one at line 2
B. The one at line 4
C. A predefined `"-"` operator for integer types
D. ***None: The code is illegal***

The `type` is used for the overload profile, and here both `T1` and `T2` are of type `T1`, which means line 4 is actually a redeclaration, which is forbidden.

Quiz

```
type T is range 0 .. 10;
subtype S is T range 1 .. 9;
```

What is the value of `S'Succ (S (9))`?

A. 9
B. 10
C. None, this fails at runtime
D. None, this does not compile

Quiz

```
type T is range 0 .. 10;
subtype S is T range 1 .. 9;
```

What is the value of `S'Succ (S (9))`?

A. 9
B. ***10***
C. None, this fails at runtime
D. None, this does not compile

`T'Succ` and `T'Pred` are defined on the **type**, not the **subtype**.

Quiz

```
type T is new Integer range 0 .. Integer'Last;
subtype S is T range 0 .. 10;

Obj : S;
```

What is the result of `Obj := S'Last + 1`?

A. 0
B. 11
C. None, this fails at runtime
D. None, this does not compile

Quiz

```
type T is new Integer range 0 .. Integer'Last;
subtype S is T range 0 .. 10;

Obj : S;
```

What is the result of `Obj := S'Last + 1`?

A. 0
B. 11
C. ***None, this fails at runtime***
D. None, this does not compile

Base Type

Base Ranges

- Actual **hardware-supported** numeric type used
  - GNAT makes consistent and predictable choices on all major platforms.
- **Predefined** operators
  - Work on full-range
    - **No range checks** on inputs or result
    - Best performance
  - Implementation may use wider registers
    - Intermediate values
- Can be accessed with `'Base` attribute

```
type Foo is range -30_000 .. 30_000;
function "+" (Left, Right : Foo'Base) return Foo'Base;
```

- Base range
  - Signed
  - 8 bits → `-128 .. 127`
  - 16 bits → `-32_768 .. 32767`

Compile-Time Constraint Violation

- *May* produce **warnings**
  - And compile successfuly
- *May* produce **errors**
  - And fail at compilation
- Requirements for rejection
  - Static value
  - Value not in range of **base** type
  - Compilation is **impossible**

```
procedure Test is
   type Some_Integer is range -200 .. 200;
   Object : Some_Integer;
begin
   Object := 50_000; -- probable error
end;
```

Range Check Failure

- Compile-time rejection
    - Depends on **base** type
    - Selected by the compiler
    - Depends on underlying **hardware**
    - Early error → "Best" case
- Else run-time **exception**
    - Most cases
    - Be happy when compilation failed instead

Real Base Decimal Precision

- Real types precision may be **better** than requested
- Example:
  - Available: 6, 12, or 24 digits of precision
  - Type with **8 digits** of precision

    ```
    type My_Type is digits 8;
    ```
  - `My_Type` will have 12 **or** 24 digits of precision

Floating Point Division By Zero

- Language-defined do as the machine does
  - If `T'Machine_Overflows` attribute is `True` raises `Constraint_Error`
  - Else $+\infty$ / $-\infty$
    - Better performance
- User-defined types always raise `Constraint_Error`

```
subtype MyFloat is Float range Float'First .. Float'Last;
type MyFloat is new Float range Float'First .. Float'Last;
```

Using Equality for Floating Point Types

- Questionable: representation issue
  - Equality $\rightarrow$ identical bits
  - Approximations $\rightarrow$ hard to **analyze**, and **not portable**
  - Related to floating-point, not Ada
- Perhaps define your own function
  - Comparison within tolerance ($+\varepsilon$ / $-\varepsilon$)

Modular Types

Bit Pattern Values and Range Constraints

- Binary based assignments possible
- No `Constraint_Error` when in range
- **Even if** they would be <= `0` as a **signed** integer type

```
procedure Demo is
  type Byte is mod 256;  -- 0 .. 255
  B : Byte;
begin
  B := 2#1000_0000#; -- not a negative value
end Demo;
```

Modular Range Must Be Respected

```
procedure P_Unsigned is
  type Byte is mod 2**8;  -- 0 .. 255
  B : Byte;
  type Signed_Byte is range -128 .. 127;
  SB : Signed_Byte;
begin
  ...
  B := -256;        -- compile error
  SB := -1;
  B := Byte (SB);   -- runtime error
  ...
end P_Unsigned;
```

Safely Converting Signed To Unsigned

- Conversion may raise `Constraint_Error`
- Use `T'Mod` to return argument `mod` `T'Modulus`
  - `Universal_Integer` argument
  - So **any** integer type allowed

```
procedure Test is
  type Byte is mod 2**8;  -- 0 .. 255
  B : Byte;
  type Signed_Byte is range -128 .. 127;
  SB : Signed_Byte;
begin
  SB := -1;
  B := Byte'Mod (SB);  -- OK (255)
```

Package **Interfaces**

- **Standard** package
- Integer types with **defined bit length**

```
type My_Base_Integer is new Integer;
pragma Assert (My_Base_Integer'First = -2**31);
pragma Assert (My_Base_Integer'Last = 2**31-1);
```

```
- Dealing with hardware registers
```

- Note: Shorter may not be faster for integer maths.
    - Modern 64-bit machines are not efficient at 8-bit maths

```
type Integer_8 is range -2**7 .. 2**7-1;
for Integer_8'Size use 8;
-- and so on for 16, 32, 64 bit types...
```

Shift/Rotate Functions

- In `Interfaces` package
  - `Shift_Left`
  - `Shift_Right`
  - `Shift_Right_Arithmetic`
  - `Rotate_Left`
  - etc.
- See RM B.2 - *The Package Interfaces*

Bit-Oriented Operations Example

- Assuming Unsigned_16 is used
  - 16-bits modular

```
with Interfaces;
use Interfaces;
...
procedure Swap( X : in out Unsigned_16 ) is
begin
  X := ( Shift_Left(X,8) and 16#FF00# ) or
       ( Shift_Right(X,8) and 16#00FF# );
end Swap;
```

Why No Implicit Shift and Rotate?

- Arithmetic, logical operators available **implicitly**
- **Why not** `Shift`, `Rotate`, etc. ?
- By **excluding** other solutions
  - As functions in **standard** → May **hide** user-defined declarations
  - As new **operators** → New operators for a **single type**
  - As **reserved words** → Not **upward compatible**

Shift/Rotate for User-Defined Types

- **Must** be modular types
- Approach 1: use `Interfaces`'s types
  - `Unsigned_8`, `Unsigned_16` ...
- Approach 2: derive from `Interfaces`'s types
  - Operations are **inherited**
  - More on that later

```
type Byte is new Interfaces.Unsigned_8;
type Half_Word is new Interfaces.Unsigned_16;
type Word is new Interfaces.Unsigned_32;
```

Quiz

```
type T is mod 256;
V : T := 255;
```

Which statement(s) is(are) legal?

A. `V := V + 1`
B. `V := 16#ff#`
C. `V := 256`
D. `V := 255 + 1`

Quiz

```
type T is mod 256;
V : T := 255;
```

Which statement(s) is(are) legal?

A. *`V := V + 1`*
B. *`V := 16#ff#`*
C. `V := 256`
D. *`V := 255 + 1`*

Quiz

```
with Interfaces; use Interfaces;

type T1 is new Unsigned_8;
V1 : T1 := 255;

type T2 is mod 256;
V2 : T2 := 255;
```

Which statement(s) is(are) legal?

A. `V1 := Rotate_Left (V1, 1)`
B. `V1 := Positive'First`
C. `V2 := 1 and V2`
D. `V2 := Rotate_Left (V2, 1)`
E. `V2 := T2'Mod (2.0)`

Quiz

```
with Interfaces; use Interfaces;

type T1 is new Unsigned_8;
V1 : T1 := 255;

type T2 is mod 256;
V2 : T2 := 255;
```

Which statement(s) is(are) legal?

A. *`V1 := Rotate_Left (V1, 1)`*
B. `V1 := Positive'First`
C. *`V2 := 1 and V2`*
D. `V2 := Rotate_Left (V2, 1)`
E. `V2 := T2'Mod (2.0)`

Representation Values

Enumeration Representation Values

- Numeric **representation** of enumerals
  - Position, unless redefined
  - Redefinition syntax

    ```
    type Enum_T is (Able, Baker, Charlie, Dog, Easy, Fox);
    for Enum_T use (1, 2, 4, 8, Easy => 16, Fox => 32);
    ```

- No manipulation *in language standard*
  - Standard is **logical** ordering
  - Ignores **representation** value
- Still accessible
  - **Unchecked** conversion
  - **Implementation**-defined facility
    - GNAT attribute `T'Enum_Rep`

Order Attributes For All Discrete Types

- **All discrete** types, mostly useful for enumerated types
- `T'Pos (Input)`
  - "Logical position number" of `Input`
- `T'Val (Input)`
  - Converts "logical position number" to `T`

```
type Days is ( Sun, Mon, Tue, Wed, Thu, Fri, Sat ); -- 0 .. 6
Today    : Days := Some_Value;
Position : Integer;
...
Position := Days'Pos( Today );
...
Get( Position );
Today := Days'Val( Position );
```

Quiz

```
type T is (Left, Top, Right, Bottom);
V : T := Left;
```

Which of the following proposition(s) are true?

A. `T'Value (V) = 1`
B. `T’Pos (V) = 1`
C. `T'Image (T'Pos (V)) = Left`
D. `T'Val (T'Pos (V) - 1) = Bottom`

Quiz

```
type T is (Left, Top, Right, Bottom);
V : T := Left;
```

Which of the following proposition(s) are true?

A. `T'Value (V) = 1`
B. ***`T'Pos (V) = 1`***
C. `T'Image (T'Pos (V)) = Left`
D. `T'Val (T'Pos (V) - 1) = Bottom`

Character Types

Language-Defined Character Types

- `Character`
  - 8-bit Latin-1
  - Base element of `String`
  - Uses attributes `'Image` / `'Value`
- `Wide_Character`
  - 16-bit Unicode
  - Base element of `Wide_Strings`
  - Uses attributes `'Wide_Image` / `'Wide_Value`
- `Wide_Wide_Character`
  - 32-bit Unicode
  - Base element of `Wide_Wide_Strings`
  - Uses attributes `'Wide_Wide_Image` / `'Wide_Wide_Value`

Character Oriented Packages

- Language-defined
- `Ada.Characters.Handling`
    - Classification
    - Conversion
- `Ada.Characters.Latin_1`
    - Characters as constants
- See RM Annex A for details

Ada.Characters.Latin_1 Sample Content

```
package Ada.Characters.Latin_1 is
  NUL : constant Character := Character'Val (0);
  ...
  LF  : constant Character := Character'Val (10);
  VT  : constant Character := Character'Val (11);
  FF  : constant Character := Character'Val (12);
  CR  : constant Character := Character'Val (13);
  ...
  Commercial_At  : constant Character := '@';  -- Character'Val(64)
  ...
  LC_A : constant Character := 'a';  -- Character'Val (97)
  LC_B : constant Character := 'b';  -- Character'Val (98)
  ...
  Inverted_Exclamation : constant Character := Character'Val (161);
  Cent_Sign            : constant Character := Character'Val (162);
...
  LC_Y_Diaeresis       : constant Character := Character'Val (255);
end Ada.Characters.Latin_1;
```

Ada.Characters.Handling Sample Content

```
package Ada.Characters.Handling is
  function Is_Control           (Item : Character) return Boolean;
  function Is_Graphic           (Item : Character) return Boolean;
  function Is_Letter            (Item : Character) return Boolean;
  function Is_Lower             (Item : Character) return Boolean;
  function Is_Upper             (Item : Character) return Boolean;
  function Is_Basic             (Item : Character) return Boolean;
  function Is_Digit             (Item : Character) return Boolean;
  function Is_Decimal_Digit     (Item : Character) return Boolean renames Is_Digit;
  function Is_Hexadecimal_Digit (Item : Character) return Boolean;
  function Is_Alphanumeric      (Item : Character) return Boolean;
  function Is_Special           (Item : Character) return Boolean;
  function To_Lower (Item : Character) return Character;
  function To_Upper (Item : Character) return Character;
  function To_Basic (Item : Character) return Character;
  function To_Lower (Item : String) return String;
  function To_Upper (Item : String) return String;
  function To_Basic (Item : String) return String;
...
end Ada.Characters.Handling;
```

Quiz

```
type T1 is (NUL = 0, A, B, 'C');
type T2 is array (Positive range <>) of T1;
Obj : T2 := "CC" & A & NUL;
```

Which of the following proposition(s) are true?

A. The code fails at runtime
B. `Obj'Length = 3`
C. Obj (1) = 'C'
D. Obj (3) = A

Quiz

```
type T1 is (NUL = 0, A, B, 'C');
type T2 is array (Positive range <>) of T1;
Obj : T2 := "CC" & A & NUL;
```

Which of the following proposition(s) are true?

A. The code fails at runtime
B. `Obj'Length = 3`
C. ***Obj (1) = 'C'***
D. ***Obj (3) = A***

Quiz

```
with Ada.Characters.Latin_1;
use Ada.Characters.Latin_1;
with Ada.Characters.Handling;
use Ada.Characters.Handling;
```

Which of the following proposition(s) are true?

A. `NUL = 0`
B. `NUL = '\0'`
C. `Character'Pos (NUL) = 0`
D. `Is_Control (NUL)`

Quiz

```
with Ada.Characters.Latin_1;
use Ada.Characters.Latin_1;
with Ada.Characters.Handling;
use Ada.Characters.Handling;
```

Which of the following proposition(s) are true?

A. `NUL = 0`
B. `NUL = '\0'`
C. *`Character'Pos (NUL) = 0`*
D. *`Is_Control (NUL)`*

Record Types

Introduction

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;
```

Components Rules

Examples

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Components_Rules is
   type File_T is record
      Name    : String (1 .. 12);
      Mode    : File_Mode;
      Size    : Integer range 0 .. 1_024;
      Is_Open : Boolean;
      -- Anonymous_Component : array (1 .. 3) of Integer;
      -- Constant_Component  : constant Integer := 123;
      -- Self_Reference      : File_T;
   end record;
   File : File_T;
begin
   File.Name    := "Filename.txt";
   File.Mode    := In_File;
   File.Size    := 0;
   File.Is_Open := False;
   Put_Line (File.Name);
end Components_Rules;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/060_record_types.html#components-rules

Characteristics of Components

- **Heterogeneous** types allowed
- Referenced **by name**
- May be no components, for **empty records**
- **No** anonymous types (e.g., arrays) allowed
- **No** constant components
- **No** recursive definitions

Components Declarations

- Multiple declarations are allowed (like objects)

```
type Several is record
   A, B, C : Integer;
end record;
```

- Recursive definitions are not allowed

```
type Not_Legal is record
  A, B : Some_Type;
  C : Not_Legal;
end record;
```

"Dot" Notation for Components 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;
```

Quiz

Which component definition is legal?

```
type Record_T is record
```

A. `Component1 : array ( 1 .. 3 ) of boolean;`
B. `Component2, Component3 :  integer;`
C. `Component4 : Record_T;`
D. `Component5 : constant integer := 123;`

```
end record;
```

Quiz

Which component definition is legal?

```
type Record_T is record
```

A. `Component1 : array ( 1 .. 3 ) of boolean;`
B. *`Component2, Component3 :  integer;`*
C. `Component4 : Record_T;`
D. `Component5 : constant integer := 123;`

```
end record;
```

Explanations

A. Anonymous types not allowed
B. Correct
C. No recursive definitions
D. No constant components

Quiz

```
type Cell is record
   Val : Integer;
   Next : Cell;
end record;
```

Is the definition legal?

A. Yes
B. No

Quiz

```
type Cell is record
   Val : Integer;
   Next : Cell;
end record;
```

Is the definition legal?

A. Yes
B. ***No***

A `record` definition cannot be recursive, here `type Cell` references itself

Quiz

```
type Cell is record
   Val : Integer;
   Message : String;
end record;
```

Is the definition legal?

A. Yes
B. No

Quiz

```
type Cell is record
   Val : Integer;
   Message : String;
end record;
```

Is the definition legal?

A. Yes
B. ***No***

A `record` definition cannot have a component of an indefinite type. `String` is indefinite if you don't specify its size.

Operations

Examples

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Operations is
   type Date_T is record
      Day   : Integer range 1 .. 31;
      Month : Positive range 1 .. 12;
      Year  : Natural range 0 .. 2_099;
   end record;
   type Personal_Information_T is record
      Name      : String (1 .. 10);
      Birthdate : Date_T;
   end record;
   type Employee_Information_T is record
      Number               : Positive;
      Personal_Information : Personal_Information_T;
   end record;
   Employee : Employee_Information_T;
begin
   Employee.Number                                := 1_234;
   Employee.Personal_Information.Name             := "Fred Smith";
   Employee.Personal_Information.Birthdate.Year := 2_020;
   Put_Line (Employee.Number'Image);
end Operations;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/060_record_types.html#operations

Available Operations

- Predefined
  - Equality (and thus inequality)

    ```
    if A = B then
    ```

  - Assignment

    ```
    A := B;
    ```

  - Component-level operations
    - Based on components' types

      ```
      if A.component < B.component then
      ```

- User-defined
  - Subprograms

Assignment Examples

```
declare
  type Complex is record
      Real : Float;
      Imaginary : Float;
    end record;
  ...
  Phase1 : Complex;
  Phase2 : Complex;
begin
  ...
    -- object reference
   Phase1 := Phase2;  -- entire object reference
   -- component references
   Phase1.Real := 2.5;
   Phase1.Real := Phase2.Real;
end;
```

Aggregates

Examples

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Aggregates is

   type Date_T is record
      Day   : Integer range 1 .. 31;
      Month : Positive range 1 .. 12;
      Year  : Natural range 0 .. 2_099;
   end record;
   type Personal_Information_T is record
      Name      : String (1 .. 10);
      Birthdate : Date_T;
   end record;
   type Employee_Information_T is record
      Number               : Positive;
      Personal_Information : Personal_Information_T;
   end record;
   Birthdate            : Date_T;
   Personal_Information : Personal_Information_T;
   Employee             : Employee_Information_T;
begin
   Birthdate := (25, 12, 2_001);
   Put_Line
     (Birthdate.Year'Image & Birthdate.Month'Image & Birthdate.Day'Image);
   Personal_Information := (Name => "Jane Smith", Birthdate => (14, 2, 2_002));
   Put_Line
     (Personal_Information.Birthdate.Year'Image &
      Personal_Information.Birthdate.Month'Image &
      Personal_Information.Birthdate.Day'Image);
   Employee := (1_234, Personal_Information => Personal_Information);
   Put_Line
     (Employee.Personal_Information.Birthdate.Year'Image &
      Employee.Personal_Information.Birthdate.Month'Image &
      Employee.Personal_Information.Birthdate.Day'Image);
   Birthdate := (Month => 1, others => 2);
   Put_Line
     (Birthdate.Year'Image & Birthdate.Month'Image & Birthdate.Day'Image);
end Aggregates;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/060_record_types.html#aggregates

Aggregates

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

```
component_init ::= expression | <>

record_aggregate ::=
   {[component_choice_list =>] component_init ,}
   [others => component_init]
```

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;
```

Aggregate Completeness

- All component values must be accounted for
  - Including defaults via `box`
- Allows compiler to check for missed components
- Type definition

```
type Struct is record
    A : Integer;
    B : Integer;
    C : Integer;
    D : Integer;
  end record;
S : Struct;
```

- Compiler will not catch the missing component

```
S.A := 10;
S.B := 20;
S.C := 12;
Send (S);
```

- Aggregate must be complete - compiler error

```
S := (10, 20, 12);
Send (S);
```

Named Associations

- **Any** order of associations
- Provides more information to the reader
  - Can mix with positional
- Restriction
  - Must stick with named associations **once started**

```
type Complex is record
    Real : Float;
    Imaginary : Float;
  end record;
Phase : Complex := (0.0, 0.0);
...
Phase := (10.0, Imaginary => 2.5);
Phase := (Imaginary => 12.5, Real => 0.212);
Phase := (Imaginary => 12.5, 0.212); -- illegal
```

Nested Aggregates

```
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;
type Person is record
   Born : Date;
   Hair : Color;
end record;
John : Person    := ( (21, November, 1990), Brown );
Julius : Person  := ( (2, August, 1995), Blond );
Heather : Person := ( (2, March, 1989), Hair => Blond );
Megan : Person   := (Hair => Blond,
                     Born => (16, December, 2001));
```

Aggregates with Only One Component

- **Must** use named form
- Same reason as array aggregates

```
type Singular is record
   A : Integer;
end record;

S : Singular := (3);          -- illegal
S : Singular := (3 + 1);      -- illegal
S : Singular := (A => 3 + 1); -- required
```

Aggregates with **others**

- Indicates all components not yet specified (like arrays)
- All `others` get the same value
  - They must be the **exact same** type

```
type Poly is record
   A : Real;
   B, C, D : Integer;
end record;

P : Poly := (2.5, 3, others => 0);

type Homogeneous is record
   A, B, C : Integer;
end record;

Q : Homogeneous := (others => 10);
```

Quiz

What is the result of running this code?

```
procedure Main is
   type Record_T is record
      A, B, C : Integer := 0;
   end record;

   V : Record_T := (A => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. Compilation error
D. Runtime error

Quiz

What is the result of running this code?

```
procedure Main is
   type Record_T is record
      A, B, C : Integer := 0;
   end record;

   V : Record_T := (A => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. ***Compilation error***
D. Runtime error

The aggregate should be written as `(A => 1, others => <>)`

Quiz

What is the result of running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer := 0;
      D : My_Integer := 0;
   end record;

   V : Record_T := (others => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. Compilation error
D. Runtime error

Quiz

What is the result of running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer := 0;
      D : My_Integer := 0;
   end record;

   V : Record_T := (others => 1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. ***Compilation error***
D. Runtime error

All components associated to a value using `others` must be of the same `type`.

Quiz

What is the result of running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer := 0;
      D : My_Integer := 0;
   end record;

   V : R := (others => <>);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. Compilation error
D. Runtime error

Quiz

What is the result of running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A, B, C : Integer := 0;
      D : My_Integer := 0;
   end record;

   V : R := (others => <>);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. *0*
B. 1
C. Compilation error
D. Runtime error

<> is an exception to the rule for **others**, it can apply to several components of a different type.

Quiz

What is the result of running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A : Integer := 0;
   end record;

   V : R := (1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. Compilation error
D. Runtime error

Quiz

What is the result of running this code?

```
procedure Main is
   type My_Integer is new Integer;
   type Record_T is record
      A : Integer := 0;
   end record;

   V : R := (1);
begin
   Put_Line (Integer'Image (V.A));
end Main;
```

A. 0
B. 1
C. ***Compilation error***
D. Runtime error

Single-valued aggregate must use named association.

Quiz

```
type Nested_T is record
   Field : Integer := 1_234;
end record;
type Record_T is record
   One   : Integer := 1;
   Two   : Character;
   Three  : Integer := -1;
   Four  : Nested_T;
end record;
X, Y : Record_T;
Z    : constant Nested_T := (others => -1);
```

Which assignment is illegal?

A. `X := (1, '2', Three => 3, Four => (6))`
B. `X := (Two => '2', Four => Z, others => 5)`
C. `X := Y`
D. `X := (1, '2', 4, (others => 5))`

Quiz

```
type Nested_T is record
   Field : Integer := 1_234;
end record;
type Record_T is record
   One    : Integer := 1;
   Two    : Character;
   Three  : Integer := -1;
   Four   : Nested_T;
end record;
X, Y : Record_T;
Z    : constant Nested_T := (others => -1);
```

Which assignment is illegal?

A. *`X := (1, '2', Three => 3, Four => (6))`*
B. `X := (Two => '2', Four => Z, others => 5)`
C. `X := Y`
D. `X := (1, '2', 4, (others => 5))`

A. Four **must** use named association
B. `others` valid: `One` and `Three` are `Integer`
C. Valid but `Two` is not initialized
D. Positional for all components

Default Values

Examples

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Default_Values is

   type Complex is record
      Real      : Float := -1.0;
      Imaginary : Float := -1.0;
   end record;

   Phasor : Complex;
   I      : constant Complex := (0.0, 1.0);

begin
   Put_Line
     (Float'Image (Phasor.Real) & " " & Float'Image (Phasor.Imaginary) & "i");
   Put_Line (Float'Image (I.Real) & " " & Float'Image (I.Imaginary) & "i");
   Phasor := (12.34, others => <>);
   Put_Line
     (Float'Image (Phasor.Real) & " " & Float'Image (Phasor.Imaginary) & "i");
end Default_Values;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/060_record_types.html#default-values

Component Default Values

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

Default Component Value Evaluation

- Occurs when object is elaborated
  - Not when the type is elaborated
- Not evaluated if explicitly overridden

```
type Structure is
  record
    A : Integer;
    R : Time := Clock;
  end record;
-- Clock is called for S1
S1 : Structure;
-- Clock is not called for S2
S2 : Structure := (A => 0, R => Yesterday);
```

Defaults Within Record Aggregates

Ada 2005

- 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 => <>);
```

Default Initialization Via Aspect Clause

Ada 2012

- Not definable for entire record type
- Components of scalar types take type's default if no explicit default value specified by record type

```
type Toggle_Switch is (Off, On)
   with Default_Value => Off;
type Controller is record
   -- Off unless specified during object initialization
   Override : Toggle_Switch;
   -- default for this component
   Enable : Toggle_Switch := On;
  end record;
C : Controller; -- Override => off, Enable => On
D : Controller := (On, Off); -- All defaults replaced
```

Quiz

Ada 2012

```
function Next return Natural; -- returns next number starting with 1

type Record_T is record
   A, B : Integer := Next;
   C    : Integer := Next;
end record;
R : Record_T := (C => 100, others => <>);
```

What is the value of R?

A. (1, 2, 3)
B. (1, 1, 100)
C. (1, 2, 100)
D. (100, 101, 102)

Quiz

Ada 2012

```
function Next return Natural; -- returns next number starting with 1

type Record_T is record
   A, B : Integer := Next;
   C    : Integer := Next;
end record;
R : Record_T := (C => 100, others => <>);
```

What is the value of R?

A. (1, 2, 3)
B. (1, 1, 100)
C. ***(1, 2, 100)***
D. (100, 101, 102)

Explanations

A. `C => 100`
B. Multiple declaration calls `Next` twice
C. Correct
D. `C => 100` has no effect on `A` and `B`

Discriminated Records

Discriminated Record Types

- **Discriminated** record type
  - Different **objects** may have **different** components
  - All object **still** share the same type
- Kind of **storage overlay**
  - Similar to `union` in C
  - But preserves **type checking**
  - And object size **depends** on discriminant
- Aggregate assignment is allowed

Discriminants

```ada
type Person_Group is (Student, Faculty);
type Person (Group : Person_Group) is record
   Name : String (1 .. 10);
   case Group is
      when Student => -- 1st variant
         Gpa  : Float range 0.0 .. 4.0;
      when Faculty => -- 2nd variant
         Pubs : Integer;
   end case;
end record;
```

- Group is the **discriminant**
- Run-time check for component **consistency**
    - eg `A_Person.Pubs := 1` checks `A_Person.Group = Faculty`
    - `Constraint_Error` if check fails
- Discriminant is **constant**
    - Unless object is **mutable**

Semantics

- Person objects are **constrained** by their discriminant
  - **Unless** mutable
  - Assignment from same variant **only**
  - **Representation** requirements

```
Pat  : Person(Student); -- No Pat.Pubs
Prof : Person(Faculty); -- No Prof.GPA
Soph : Person := ( Group  => Student,
                   Name => "John Jones",
                   GPA  => 3.2);
X : Person;  -- Illegal: must specify discriminant
```

```
Pat  := Soph; -- OK
Soph := Prof; -- Constraint_Error at run time
```

Mutable Discriminated Record

- When discriminant has a **default value**
  - Objects instantiated **using the default** are **mutable**
  - Objects specifying an **explicit** value are **not** mutable
- Mutable records have **variable** discriminants
- Use **same** storage for **several** variant

```
-- Potentially mutable
type Person (Group : Person_Group := Student) is record

--  Use default value: mutable
S : Person;
--  Explicit value: *not* mutable
--  even if Student is also the default
S2 : Person (Group => Student);
...
S := (Group => Student, Gpa => 0.0);
S := (Group => Faculty, Pubs => 10);
```

Lab

Record Types Lab

- Requirements
  - Create a simple First-In/First-Out (FIFO) queue record type and object
  - Allow the user to:
    - Add ("push") items to the queue
    - Remove ("pop") the next item to be serviced from the queue (Print this item to ensure the order is correct)
  - When the user is done manipulating the queue, print out the remaining items in the queue
- Hints
  - Queue record should at least contain:
    - Array of items
    - Index into array where next item will be added

Record Types Lab Solution - Declarations

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is

   type Name_T is array (1 .. 6) of Character;
   type Index_T is range 0 .. 1_000;
   type Queue_T is array (Index_T range 1 .. 1_000) of Name_T;

   type Fifo_Queue_T is record
      Next_Available : Index_T := 1;
      Last_Served    : Index_T := 0;
      Queue          : Queue_T := (others => (others => ' '));
   end record;

   Queue : Fifo_Queue_T;
   Choice : Integer;
```

Record Types Lab Solution - Implementation

```
begin

   loop
      Put ("1 = add to queue | 2 = remove from queue | others => done: ");
      Choice := Integer'Value (Get_Line);
      if Choice = 1 then
         Put ("Enter name: ");
         Queue.Queue (Queue.Next_Available) := Name_T (Get_Line);
         Queue.Next_Available               := Queue.Next_Available + 1;
      elsif Choice = 2 then
         if Queue.Next_Available = 1 then
            Put_Line ("Nobody in line");
         else
            Queue.Last_Served := Queue.Last_Served + 1;
            Put_Line ("Now serving: " & String (Queue.Queue (Queue.Last_Served)));
         end if;
      else
         exit;
      end if;
      New_Line;
   end loop;

   Put_Line ("Remaining in line: ");
   for Index in Queue.Last_Served + 1 .. Queue.Next_Available - 1 loop
      Put_Line ("  " & String (Queue.Queue (Index)));
   end loop;

end Main;
```

Summary

Summary

- Heterogeneous types allowed for components
- Default initial values allowed for components
  - Evaluated when each object elaborated, not the type
  - Not evaluated if explicit initial value specified
- Aggregates express literals for composite types
  - Can mix named and positional forms

Discriminated Record Types

Introduction

Variant Record Types

- *Variant record type* is a record type where
  - Different objects may have different sets of components (i.e. different variants)
  - Given object itself may be *unconstrained*
    - Different variants at different times
- Supported in other languages
  - Variant records in Pascal
  - Unions in C
- Variant record offers a kind of storage overlaying
  - Same storage might be used for one variant at one time, and then for another variant later
  - Language issue: Ensure this does not provide loophole from type checking
    - Neither Pascal nor C avoids this loophole

Example Variant Record Description

- Record / structure type for a person
  - Person is either a student or a faculty member (discriminant)
  - Person has a name (string)
  - Each student has a GPA (floating point) and a graduation year (non-negative integer)
  - Each faculty has a count of publications (non-negative integer)

Example Defined in C

```c
enum person_tag {Student, Faculty};

struct Person {
   enum person_tag tag;
   char name [10];
   union {
      struct { float gpa; int year; } s;
      int pubs;
   };
};
```

- Issue: maintaining consistency between tag and union fields is responsibility of the programmer
  - Source of potential vulnerabilities

Example Defined in Ada

```
type Person_Tag is (Student, Faculty);
type Person (Tag : Person_Tag) is -- Tag is the discriminant
   record
      Name : String(1..10); -- Always present
      case Tag is
         when Student => -- 1st variant
            GPA  : Float range 0.0 .. 4.0;
            Year : Integer range 1..4;
         when Faculty => -- 2nd variant
            Pubs : Integer;
      end case;
   end record;
```

- **Tag** value enforces field availability
  - Can only access **GPA** and **Year** when **Tag** is **Student**
  - Can only access **Pubs** when **Tag** is **Faculty**

Variant Part of Record

- Variant part of record specifies alternate list of componenents

```
type Variant_Record_T (Discriminant : Integer) is record
   Common_Component : String (1 .. 10);
   case Discriminant is
      when Integer'First .. -1 =>
         Negative_Component : Float;
      when 1 .. Integer'Last =>
         Positive_Component : Integer;
      when others =>
         Zero_Component : Boolean;
   end case;
end record;
```

  - Choice is determined by discriminant value
  - Record can only contain one variant part
    - Variant must be last part of record definition

Variant Record Semantics

Discriminant in Ada Variant Records

- Variant record type contains a special field (*discriminant*) whose value indicates which variant is present
- When a field in a variant is selected, run-time check ensures that discriminant value is consistent with the selection
    - If you could store into **Pubs** but read **GPA**, type safety would not be guaranteed
- Ada prevents this type of access
    - Discriminant (Tag) established when object of type Person created
    - Run-time check verifies that field selected from variant is consistent with discriminant value
        - Constraint_Error raised if the check fails
- Can only read discriminant (as any other field), not write
    - Aggregate assignment is allowed

Semantics

- Variable of type **Person** is constrained by value of discriminant supplied at object declaration
  - Determines minimal storage requirements
  - Limits object to corresponding variant

```
Pat  : Person(Student); -- May select Pat.GPA, not Pat.Pubs
Prof : Person(Faculty); -- May select Prof.Pubs, not Prof.GPA
Soph : Person := ( Tag  => Student,
                   Name => "John Jones",
                   GPA  => 3.2,
                   Year => 2);
X    : Person;  -- Illegal; discriminant must be initialized
```

- Assignment between Person objects requires same discriminant values for LHS and RHS

```
Pat  := Soph; -- OK
Soph := Prof; -- Constraint_Error at run time
```

Implementation

- Typically type and operations would be treated as an ADT
  - Implemented in its own package

```
package Person_Pkg is
   type Person_Tag is (Student, Faculty);
   type Person (Tag : Person_Tag) is
      record
         Name : String(1..10);
         case Tag is
            when Student =>
               GPA  : Float range 0.0 .. 4.0;
               Year : Integer range 1..4;
            when Faculty =>
               Pubs : Integer;
         end case;
      end record;
   -- parameters can be unconstrained (constraint comes from caller)
   procedure Put ( Item : in Person );
   procedure Get ( Item : in out Person );
end Person_Pkg;
```

Primitives

- Output

```
procedure Put ( Item : in Person ) is
begin
  Put_Line("Tag:" & Person_Tag'Image(Item.Tag));
  Put_Line("Name: " & Item.Name );
  -- Tag specified by caller
  case Item.Tag is
    when Student =>
      Put_Line("GPA:" & Float'Image(Item.GPA));
      Put_Line("Year:" & Integer'Image(Item.Year) );
    when Faculty =>
      Put_Line("Pubs:" & Integer'Image(Item.Pubs) );
  end case;
end Put;
```

- Input

```
procedure Get ( Item : in out Person ) is
begin
  -- Tag specified by caller
  case Item.Tag is
    when Student =>
      Item.GPA := Get_GPA;
      Item.Year := Get_Year;
    when Faculty =>
      Item.Pubs := Get_Pubs;
  end case;
end Get;
```

Usage

```
with Person_Pkg; use Person_Pkg;
with Ada.Text_IO; use Ada.Text_IO;
procedure Person_Test is
  Tag   : Person_Tag;
  Line  : String(1..80);
  Index : Natural;
begin
  loop
    Put("Tag (Student or Faculty, empty line to quit): ");
    Get_Line(Line, Index);
    exit when Index=0;
    Tag := Person_Tag'Value(Line(1..Index));
    declare
      Someone : Person(Tag);
    begin
      Get(Someone);
      case Someone.Tag is
        when Student => Student_Do_Something ( Someone );
        when Faculty => Faculty_Do_Something ( Someone );
      end case;
      Put(Someone);
    end;
  end loop;
end Person_Test;
```

Unconstrained Variant Records

Adding Flexibility to Variant Records

- Previously, declaration of **Person** implies that object, once created, is always constrained by initial value of **Tag**
    - Assigning **Person(Faculty)** to **Person(Student)** or vice versa, raises Constraint_Error
- Additional flexibility is sometimes desired
    - Allow declaration of unconstrained **Person**, to which either **Person(Faculty)** or **Person(Student)** can be assigned
    - To do this, *declare discriminant with default initialization*
- Type safety is not compromised
    - Modification of discriminant is only permitted when entire record is assigned
        - Either through copying an object or aggregate assignment

Unconstrained Variant Record Example

```
declare
   type Mutant( Tag : Person_Tag := Faculty ) is
      record
         Name : String(1..10);
         case Tag is
            when Student =>
               GPA  : Float range 0.0 .. 4.0;
               Year : Integer range 1..4;
            when Faculty =>
               Pubs : Integer;
         end case;
      end record;

   Pat  : Mutant( Student ); -- Constrained
   Doc  : Mutant( Faculty ); -- Constrained
   Zork : Mutant; -- Unconstrained (Zork.Tag = Faculty)

begin
   Zork     := Pat;      -- OK, Zork.Tag was Faculty, is now Student
   Zork.Tag := Faculty; -- Illegal to assign to discriminant
   Zork     := Doc;      -- OK, Zork.Tag is now Faculty
   Pat      := Zork;     -- Run-time error (Constraint_Error)
end;
```

Quiz

```
procedure Main is
   type Shape_Kind is (Circle, Line);

   type Shape (Kind : Shape_Kind) is record
      case Kind is
         when Line =>
            X, Y : Float;
            X2, Y2 : Float;
         when Circle =>
            Radius : Float;
      end case;
   end record;
begin
   V := V2;
```

Which declaration(s) is(are) legal for this piece of code?

A.
```
V : Shape := (Circle, others => 0.0)
V2 : Shape (Line);
```

B.
```
V : Shape := (Kind => Circle, Radius => 0.0);
V2 :  Shape (Circle);
```

C.
```
V : Shape (Line) := (Kind => Circle, Radius => 0.0);
V2 : Shape (Circle);
```

D.
```
V : Shape;
V2 : Shape (Circle);
```

Quiz

```
procedure Main is
   type Shape_Kind is (Circle, Line);

   type Shape (Kind : Shape_Kind) is record
      case Kind is
         when Line =>
            X, Y : Float;
            X2, Y2 : Float;
         when Circle =>
            Radius : Float;
      end case;
   end record;
begin
   V := V2;
```

Which declaration(s) is(are) legal for this piece of code?

A. `V : Shape := (Circle, others => 0.0)`
   `V2 : Shape (Line);`

B. *`V : Shape := (Kind => Circle, Radius => 0.0);`*
   `V2 :  Shape (Circle);`

C. `V : Shape (Line) := (Kind => Circle, Radius => 0.0);`
   `V2 : Shape (Circle);`

D. `V : Shape;`
   `V2 : Shape (Circle);`

A. Cannot assign with different discriminant
B. OK
C. `V` initial value has a different discriminant
D. `Shape` cannot be mutable: `V` must have a discriminant

Quiz

```
type Shape_Kind is (Circle, Line);

type Shape (Kind : Shape_Kind) is record
   case Kind is
      when Line =>
         X, Y : Float;
         X2, Y2 : Float;
```

Which declaration(s) is(are) legal?

A.
```
when Circle =>
   Cord : Shape (Line);
```

B.
```
when Circle =>
   Center : array (1 .. 2) of Float;
   Radius : Float;
```

C.
```
when Circle =>
   Center_X, Center_Y : Float;
   Radius :  Float;
```

D.
```
when Circle =>
   X, Y, Radius : Float;
```

Quiz

```
type Shape_Kind is (Circle, Line);

type Shape (Kind : Shape_Kind) is record
   case Kind is
      when Line =>
         X, Y : Float;
         X2, Y2 : Float;
```

Which declaration(s) is(are) legal?

A.
```
when Circle =>
   Cord : Shape (Line);
```

B.
```
when Circle =>
   Center : array (1 .. 2) of Float;
   Radius : Float;
```

C.
```
when Circle =>
   Center_X, Center_Y : Float;
   Radius :  Float;
```

D.
```
when Circle =>
   X, Y, Radius : Float;
```

A. Referencing itself
B. anonymous array in record declaration
C. OK
D. X, Y are duplicated with the Line variant

Varying Length Arrays

Varying Lengths of Array Objects

- In Ada, array objects have to be fixed length

```
S : String(1..80);
A : array ( M .. K*L ) of Integer;
```

- We would like an object with a maximum length, but current length is variable
  - Need two pieces of data
    - Array contents
    - Location of last valid element
- For common usage, we want this to be a type (probably a record)
  - Maximum size array for contents
  - Index for last valid element

Simple Varying Length Array

```
type Simple_VString is
   record
      Length : Natural range 0..Max_Length := 0;
      Data   : String(1..Max_Length) := (others => ' ');
   end record;

function "&"(Left, Right : Simple_VString) return Simple_VString is
   Result : Simple_VString;
begin
   if Left.Length + Right.Length > Max_Length then
      raise Constraint_Error;
   else
      Result.Length := Left.Length + Right.Length;
      Result.Data(1..Result.Length) :=
         Left.Data(1..Left.Length) & Right.Data(1..Right.Length);
      return Result;
   end if;
end "&";
```

- Issues
  - Every object has same maximum length
  - **Length** needs to be maintained by program logic
  - Need to define "="

Varying Length Array via Variant Records

- Discriminant can serve as bound of array field

```
type VString ( Max_Length : Natural := 0 ) is
  record
    Data   : String(1..Max_Length) := (others => ' ');
  end record;
```

- Discriminant default value?
  - With default discriminant value, objects can be copied even if lengths are different
  - With no default discriminant value, objects of different lengths cannot be copied

Quiz

```
type My_Array is array (Integer range <>) of Boolean;
```

How to declare an array of two elements?

A. `O : My_Array (2)`
B. `O : My_Array (1 .. 2)`
C. `O : My_Array (1 .. 3)`
D. `O : My_Array (1, 3)`

Quiz

```
type My_Array is array (Integer range <>) of Boolean;
```

How to declare an array of two elements?

A. `O : My_Array (2)`
B. *`O : My_Array (1 .. 2)`*
C. `O : My_Array (1 .. 3)`
D. `O : My_Array (1, 3)`

Quiz

```
type R (Size : Integer := 0) is record
   S : String (1 .. Size);
end record;
```

Which proposition(s) will compile and run without error?

A. `V : R := (6, "Hello")`
B. `V : R := (5, "Hello")`
C. `V : R (5) := (5, S => "Hello")`
D. `V : R (6) := (6, S => "Hello")`

Quiz

```
type R (Size : Integer := 0) is record
   S : String (1 .. Size);
end record;
```

Which proposition(s) will compile and run without error?

A. `V : R := (6, "Hello")`
B. `V : R := (5, "Hello")`
C. ***`V : R (5) := (5, S => "Hello")`***
D. `V : R (6) := (6, S => "Hello")`

When V is declared without specifying its size, it becomes mutable, at this point the `S'Length = Positive'Last`, causing a Runtime_Error. Furthermore the length of "Hello" is 5, it cannot be stored in a String of Length 6.

Variant Record Details

Semantics of Discriminated Records

- A discriminant is a parameter to a record type
  - The value of a discriminant affects the presence, constraints, or initialization of other components
- A type may have more than one discriminant
  - Either all have default initializations, or none do
- Ada restricts the kinds of types that may be used to declare a discriminant
  - Discrete types (i.e., enumeration or integer type)
  - Access types (not covered here)

Use of Discriminants in Record Definition

- Within the record type definition, a discriminant may only be referenced in the following contexts
  - In "case" of variant part
  - As a bound of a record component that is an unconstrained array
  - As an initialization expression for a component
  - As the value of a discriminant for a component that itself a variant record
- A discriminant is not allowed as the bound of a range constraint

Lab

Discriminated Record Types Lab

- Requirements for a simplistic employee database
  - Create a package to handle varying length strings using variant records
    - The string type **must** be **private**!
    - The variant can appear on the partial definition or the full
  - Create a package to create employee data in a variant record
    - Store first name, last name, and hourly pay rate for all employees
    - Supervisors must also include the project they are supervising
    - Managers must also include the number of employees they are managing and the department name
  - Main program should read employee information from the console
    - Any number of any type of employees can be entered in any order
    - When data entry is done, print out all appropriate information for each employee
- Hints
  - Create concatenation functions for your varying length string type
  - Is it easier to create an input function for each employee category, or a common one?

Discriminated Record Types Lab Solution - Vstring

```
package Vstring is
   Max_String_Length : constant := 1_000;
   type Vstring_T is private;
   function To_Vstring (Str : String) return Vstring_T;
   function To_String (Vstr : Vstring_T) return String;
   function "&" (L, R : Vstring_T) return Vstring_T;
   function "&" (L : String; R : Vstring_T) return Vstring_T;
   function "&" (L : Vstring_T; R : String) return Vstring_T;
private
   subtype Index_T is Integer range 0 .. Max_String_Length;
   type Vstring_T (Length : Index_T := 0) is record
      Text : String (1 .. Length);
   end record;
end Vstring;

package body Vstring is
   function To_Vstring (Str : String) return Vstring_T is
      ((Length => Str'Length, Text => Str));
   function To_String (Vstr : Vstring_T) return String is
      (Vstr.Text);
   function "&" (L, R : Vstring_T) return Vstring_T is
      Ret_Val : constant String := L.Text & R.Text;
   begin
      return (Length => Ret_Val'Length, Text => Ret_Val);
   end "&";

   function "&" (L : String; R : Vstring_T) return Vstring_T is
      Ret_Val : constant String := L & R.Text;
   begin
      return (Length => Ret_Val'Length, Text => Ret_Val);
   end "&";

   function "&" (L : Vstring_T; R : String) return Vstring_T is
      Ret_Val : constant String := L.Text & R;
   begin
      return (Length => Ret_Val'Length, Text => Ret_Val);
   end "&";
end Vstring;
```

Discriminated Record Types Lab Solution - Employee (Spec)

```
with Vstring;     use Vstring;
package Employee is

   type Category_T is (Staff, Supervisor, Manager);
   type Pay_T is delta 0.01 range 0.0 .. 1_000.00;

   type Employee_T (Category : Category_T := Staff) is record
      Last_Name   : Vstring.Vstring_T;
      First_Name  : Vstring.Vstring_T;
      Hourly_Rate : Pay_T;
      case Category is
         when Staff =>
            null;
         when Supervisor =>
            Project : Vstring.Vstring_T;
         when Manager =>
            Department  : Vstring.Vstring_T;
            Staff_Count : Natural;
      end case;
   end record;

   function Get_Staff return Employee_T;
   function Get_Supervisor return Employee_T;
   function Get_Manager return Employee_T;

end Employee;
```

Discriminated Record Types Lab Solution - Employee (Body)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Employee is
   function Read (Prompt : String) return String is
   begin
      Put (Prompt & " > ");
      return Get_Line;
   end Read;

   function Get_Staff return Employee_T is
      Ret_Val : Employee_T (Staff);
   begin
      Ret_Val.Last_Name   := To_Vstring (Read ("Last name"));
      Ret_Val.First_Name  := To_Vstring (Read ("First name"));
      Ret_Val.Hourly_Rate := Pay_T'Value (Read ("Hourly rate"));
      return Ret_Val;
   end Get_Staff;

   function Get_Supervisor return Employee_T is
      Ret_Val : Employee_T (Supervisor);
   begin
      Ret_Val.Last_Name   := To_Vstring (Read ("Last name"));
      Ret_Val.First_Name  := To_Vstring (Read ("First name"));
      Ret_Val.Hourly_Rate := Pay_T'Value (Read ("Hourly rate"));
      Ret_Val.Project     := To_Vstring (Read ("Project"));
      return Ret_Val;
   end Get_Supervisor;

   function Get_Manager return Employee_T is
      Ret_Val : Employee_T (Manager);
   begin
      Ret_Val.Last_Name   := To_Vstring (Read ("Last name"));
      Ret_Val.First_Name  := To_Vstring (Read ("First name"));
      Ret_Val.Hourly_Rate := Pay_T'Value (Read ("Hourly rate"));
      Ret_Val.Department  := To_Vstring (Read ("Department"));
      Ret_Val.Staff_Count := Integer'Value (Read ("Staff count"));
      return Ret_Val;
   end Get_Manager;
end Employee;
```

Discriminated Record Types Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Employee;
with Vstring; use Vstring;
procedure Main is
   procedure Print (Member : Employee.Employee_T) is
      First_Line : constant Vstring.Vstring_T :=
        Member.First_Name & " " & Member.Last_Name & " " &
        Member.Hourly_Rate'Image;
   begin
      Put_Line (Vstring.To_String (First_Line));
      case Member.Category is
         when Employee.Supervisor =>
            Put_Line ("   Project: " & Vstring.To_String (Member.Project));
         when Employee.Manager =>
            Put_Line ("   Overseeing " & Member.Staff_Count'Image & " in " &
                      Vstring.To_String (Member.Department));
         when others => null;
      end case;
   end Print;

   List  : array (1 .. 1_000) of Employee.Employee_T;
   Count : Natural := 0;
begin
   loop
      Put_Line ("E => Employee");
      Put_Line ("S => Supervisor");
      Put_Line ("M => Manager");
      Put ("E/S/M (any other to stop): ");
      declare
         Choice : constant String := Get_Line;
      begin
         case Choice (1) is
            when 'E' | 'e' =>
               Count        := Count + 1;
               List (Count) := Employee.Get_Staff;
            when 'S' | 's' =>
               Count        := Count + 1;
               List (Count) := Employee.Get_Supervisor;
            when 'M' | 'm' =>
               Count        := Count + 1;
               List (Count) := Employee.Get_Manager;
            when others =>
               exit;
         end case;
      end;
   end loop;

   for Item of List (1 .. Count) loop
      Print (Item);
   end loop;
end Main;
```

Summary

Properties of Variant Record Types

- Rules
  - Case choices for variants must partition possible values for discriminant
  - Field names must be unique across all variants
- Style
  - Typical processing is via a case statement that "dispatches" based on discriminant
  - This centralized functional processing is in contrast to decentralized object-oriented approach
- Flexibility
  - Variant parts may be nested, if some fields common to a set of variants

Advanced Primitives

Type Derivation

Freeze Point

- Ada doesn't explicitly identify the end of members declaration
- This end is the implicit **freeze point** occurring whenever:
  - A **variable** of the type is **declared**
  - The type is **derived**
  - The **end of the scope** is reached
- Subprograms past this point are not primitive

```
type Root is Integer;
procedure Prim (V : Root);
type Child is new Root; -- freeze root
procedure Prim2 (V : Root); -- Not a primitive

V : Child; --  freeze child
procedure Prim3 (V : Child); -- Not a primitive
```

Primitive of Multiple Types

- A subprogram can be a primitive of several types

```
package P is
   type T1 is range 1 .. 10;
   type T2 is (A, B, C);

   procedure Proc (V1 : T1; V2 : T2);
   function "+" (V1 : T1; V2 : T2) return T1;
end P;
```

Implicit Primitive Operations

- Type declaration implicitly creates primitives
  - Numerical and logical operations
  - Code can overload or remove them

```
package P is
   type T1 is range 1 .. 10;
   -- implicit
   -- function "+" (Left, Right : T1) return T1;
end P;
...
procedure Main is
   V1, V2 : T1;
begin
   V1 := V1 + V2;
end Main;
```

Recap. on type derivation

- For all types
    - Freeze point rules don't change
    - Primitives are inherited by child types
    - Conversion from child to parent possible
    - Pre-defined set of primitives
        - `"+", "-" ...` for numeric types
        - Comparison operators
        - Equality except if `limited`
- Derived type that are **not** `tagged`
    - Are **not** OOP
    - Can remove a primitive
    - Can declare a primitive of multiple types
    - Can be converted from parent to child
        - Their representation does not change
        - Could raise `Constraint_Error` (`range`...)

Quiz

```
type T is new Integer;
```

Which operator(s) definition(s) is legal?

A. `function "+" (V : T) return Boolean is (T /= 0)`
B. `function "+" (A, B : T) return T is (A + B)`
C. `function "=" (A, B : T) return T is (A - B)`
D. `function ":=" (A : T) return T is (A)`

Quiz

```
type T is new Integer;
```

Which operator(s) definition(s) is legal?

A. `function "+" (V : T) return Boolean is (T /= 0)`
B. *`function "+" (A, B : T) return T is (A + B)`*
C. *`function "=" (A, B : T) return T is (A - B)`*
D. `function ":=" (A : T) return T is (A)`

B. Infinite recursion
C. Unlike some languages, there is no assignment operator

Quiz

```
type T1 is new Integer;
function "+" (A : T1) return T1 is (0);
type T2 is new T1;
type T3 is new T1;
overriding function "+" (A : T3) return T3 is (1);

O1 : T1;
O2 : T2;
O3 : T3;
```

Which proposition(s) is(are) legal and running without error?

A. `pragma Assert (+O1 = 0)`
B. `pragma Assert (+O2 = 0)`
C. `pragma Assert ((+O2) + (+O3) = 1)`
D. `pragma Assert (+(T3 (O1) + O3) = 1)`

Quiz

```ada
type T1 is new Integer;
function "+" (A : T1) return T1 is (0);
type T2 is new T1;
type T3 is new T1;
overriding function "+" (A : T3) return T3 is (1);

O1 : T1;
O2 : T2;
O3 : T3;
```

Which proposition(s) is(are) legal and running without error?

A. *pragma Assert (+O1 = 0)*
B. *pragma Assert (+O2 = 0)*
C. pragma Assert ((+O2) + (+O3) = 1)
D. *pragma Assert (+(T3 (O1) + O3) = 1)*

C. +O2 returns a T2, +O3 a T3

Tagged Inheritance

Liskov's Substitution Principle

- *LSP* is an object-oriented rule
  - Not imposed
    - But fits nicely with Ada's OOP design
  - Avoids numerous issues
  - Can be verified by tools eg. CodePeer
- *Objects of a parent type shall be replaceable by objects of its child types*
  - Cannot be applied to simple derivation (eg. restricting range)
  - Tagged record derivation implies **extending** not modifying the behaviour
    - Easier said than done
    - *Is a mute cat still a cat if it can't meow?*

Dispatching

- Primitives dispatch, but not only them

```
type T is tagged null record;

procedure Prim (A : T) is null;
procedure Not_Prim (A : T'Class) is null;
```

- `Prim` is a primitive
- `Not_Prim` is **not** a primitive
  - Won't be inherited
  - But dispatches dynamically!

```
declare
    type T2 is new T with null record;
    A : T'Class := T2'(null record);
begin
    Prim (A);
    Not_Prim (A);
end;
```

Tagged Primitive Declaration

- **tagged** types primitives **must** be declared in a **package** specification
- Not a **declare** block or the declarative part of a subprogram

```
procedure P is
   type T is tagged null record;
   procedure Not_Prim (A : T) is null;
   type T2 is tagged null record;

   A : T;
   B : T2;
begin
   -- Not a primitive
   Not_Prim (A);
   -- Would not compile
   -- Not_Prim (B);
end P;
```

Primitive of Multiple Types

- For a primitive of a `tagged record` `Tag_T`
  - `Tag_T` is called the *controlling parameter*
  - All controlling parameters **must** be of the same type
- Warning: A non-tagged type is never controlling
  - Can have primitive of multiple `type`
  - **Cannot** have primitive of multiple `tagged record`

```
type Root1 is tagged null record;
type Root2 is tagged null record;

procedure P1_Correct (V1 : Root1; V2 : Root1);
procedure P2_Incorrect (V1 : Root1; V2 : Root2); -- FAIL
```

Recap. on tagged inheritance

- `tagged` types are Ada's OOP
- They can
  - Be converted from a parent: `Child_Type (Parent)`
- They **cannot**
  - Remove a primitive
  - Have a primitive with multiple controlling types
  - Be converted to a parent: `Parent_Type (Child)`
    - Because their representation may change

Quiz

```
type T1 is range 0 .. 10;
type T2 is range 0 .. 10;
type Tag_T1 is tagged null record;
type Tag_T2 is tagged null record;
```

Which of the following piece(s) of code is(are) legal?

A. `procedure P (A : T1; B : T2) is null`
B. `procedure P (A : T1; B : Tag_T1) is null`
C. `procedure P (A : T1; B : Tag_T1; C : Tag_T1) is null`
D. `procedure P (A : T1; B : Tag_T1; C : Tag_T2) is null`

Quiz

```
type T1 is range 0 .. 10;
type T2 is range 0 .. 10;
type Tag_T1 is tagged null record;
type Tag_T2 is tagged null record;
```

Which of the following piece(s) of code is(are) legal?

A. *procedure P (A : T1; B : T2) is null*
B. *procedure P (A : T1; B : Tag_T1) is null*
C. *procedure P (A : T1; B : Tag_T1; C : Tag_T1) is null*
D. `procedure P (A : T1; B : Tag_T1; C : Tag_T2) is null`

D. Has two controlling type

Quiz

```
type T1 is tagged null record;
type T2 is new T1 with null record;
V : T2;
```

Which of the following piece(s) of code allow for calling `Proc (V)`?

A. `procedure Proc (V : T1) is null`

B. `procedure Proc (V : T1’Class) is null`

C.
```
procedure Proc (V : T1'Class) is null;
procedure Proc (V : T2'Class) is null;
```

D.
```
procedure Proc (V : T1) is null;
procedure Proc (V : T2) is null;
```

Quiz

```
type T1 is tagged null record;
type T2 is new T1 with null record;
V : T2;
```

Which of the following piece(s) of code allow for calling `Proc (V)`?

A. `procedure Proc (V : T1) is null`

B. *`procedure Proc (V : T1'Class) is null`*

C.
```
procedure Proc (V : T1'Class) is null;
procedure Proc (V : T2'Class) is null;
```

D.
```
procedure Proc (V : T1) is null;
procedure Proc (V : T2) is null;
```

A. Proc is not a primitive

B. `T1'Class` contains `T2'Class`

Quantified Expressions

Quantified Expressions

Introduction

Ada 2012

- Expressions that have a Boolean value
- The value indicates something about a set of objects
    - In particular, whether something is True about that set
- That "something" is expressed as an arbitrary boolean expression
    - A so-called "predicate"
- "Universal" quantified expressions
    - Indicate whether predicate holds for all components
- "Existential" quantified expressions
    - Indicate whether predicate holds for at least one component

Examples

```
with GNAT.Random_Numbers; use GNAT.Random_Numbers;
with Ada.Text_IO;         use Ada.Text_IO;
procedure Quantified_Expressions is
   Gen    : Generator;
   Values : constant array (1 .. 10) of Integer := (others => Random (Gen));

   Any_Even : constant Boolean := (for some N of Values => N mod 2 = 0);
   All_Odd  : constant Boolean := (for all N of reverse Values => N mod 2 = 1);

   function Is_Sorted return Boolean is
     (for all K in Values'Range =>
        K = Values'First or else Values (K - 1) <= Values (K));

   function Duplicate return Boolean is
     (for some I in Values'Range =>
        (for some J in I + 1 .. Values'Last => Values (I) = Values (J)));

begin
   Put_Line ("Any Even: " & Boolean'Image (Any_Even));
   Put_Line ("All Odd: " & Boolean'Image (All_Odd));
   Put_Line ("Is_Sorted " & Boolean'Image (Is_Sorted));
   Put_Line ("Duplicate " & Boolean'Image (Duplicate));
end Quantified_Expressions;
```

Semantics Are As If You Wrote This Code

Ada 2012

```
function Universal (Set : Components) return Boolean is
begin
  for C of Set loop
    if not Predicate (C) then
      return False;  -- Predicate must be true for all
    end if;
  end loop;
  return True;
end Universal;

function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True;  -- Predicate need only be true for one
    end if;
  end loop;
  return False;
end Existential;
```

Quantified Expressions Syntax

Ada 2012

```
quantified_expression ::=
   (for quantifier loop_parameter_specification
         => predicate) |
   (for quantifier iterator_specification =>
         predicate)
predicate ::= boolean_expression
quantifier ::= all | some
```

Simple Examples

Ada 2012

```
Values : constant array (1 .. 10) of Integer := (...);
Is_Any_Even : constant Boolean :=
   (for some V of Values => V mod 2 = 0);
Are_All_Even : constant Boolean :=
   (for all V of Values => V mod 2 = 0);
```

Universal Quantifier

Ada 2012

- In logic, denoted by ∀ (inverted 'A', for "all")
- "There is no member of the set for which the predicate does not hold"
  - If predicate is False for any member, the whole is False
- Functional equivalent

```
function Universal (Set : Components) return Boolean is
begin
  for C of Set loop
    if not Predicate (C) then
       return False; -- Predicate must be true for all
    end if;
  end loop;
  return True;
end Universal;
```

Universal Quantifier Illustration

Ada 2012

- "There is no member of the set for which the predicate does not hold"
- Given a set of integer answers to a quiz, there are no answers that are not 42 (i.e., all are 42)

```
Ultimate_Answer : constant := 42; -- to everything...
Answers : constant array (1 .. 10)
    of Integer := ( ... );
All_Correct_1 : constant Boolean :=
   (for all Component of Answers =>
      Component = Ultimate_Answer);
All_Correct_2 : constant Boolean :=
   (for all K in Answers'range =>
      Answers(K) = Ultimate_Answer);
```

Universal Quantifier Real-World Example

Ada 2012

```
type DMA_Status_Flag is ( ... );
function Status_Indicated (
  Flag : DMA_Status_Flag)
  return Boolean;
None_Set : constant Boolean := (
  for all Flag in DMA_Status_Flag =>
    not Status_Indicated (Flag));
```

Existential Quantifier

Ada 2012

- In logic, denoted by ∃ (rotated 'E', for "exists")
- "There is at least one member of the set for which the predicate holds"
  - If predicate is True for any member, the whole is True
- Functional equivalent

```
function Existential (Set : Components) return Boolean is
begin
  for C of Set loop
    if Predicate (C) then
      return True; -- Need only be true for at least one
    end if;
  end loop;
  return False;
end Existential;
```

Existential Quantifier Illustration

Ada 2012

- "There is at least one member of the set for which the predicate holds"
- Given set of integer answers to a quiz, there is at least one answer that is 42

```
Ultimate_Answer : constant := 42; -- to everything...
Answers : constant array (1 .. 10)
    of Integer := ( ... );
Any_Correct_1 : constant Boolean :=
   (for some Component of Answers =>
      Component = Ultimate_Answer);
Any_Correct_2 : constant Boolean :=
   (for some K in Answers'range =>
      Answers(K) = Ultimate_Answer);
```

Index-Based vs Component-Based Indexing

Ada 2012

- Given an array of integers

```
Values : constant array (1 .. 10) of Integer := (...);
```

- Component-based indexing is useful for checking individual values

```
Contains_Negative_Number : constant Boolean :=
   (for some N of Values => N < 0);
```

- Index-based indexing is useful for comparing across values

```
Is_Sorted : constant Boolean :=
   (for all I in Values'Range =>
      I = Values'first or else Values(I) >= Values(I-1))
```

"Pop Quiz" for Quantified Expressions

Ada 2012

- What will be the value of **Ascending_Order**?

```
Table : constant array (1 .. 10) of Integer :=
      (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Ascending_Order : constant Boolean := (
  for all K in Table'Range =>
    K > Table'First and then Table (K - 1) <= Table (K))
```

  - Answer: **False**. Predicate fails when **K = Table'First**
    - First subcondition is False!
    - Condition should be

```
Ascending_Order : constant Boolean := (
   for all K in Table'Range => K = Table'first or else
                               Table (K - 1) <= Table (K));
```

When The Set Is Empty...

Ada 2012

- Universally quantified expressions are True
  - Definition: there is no member of the set for which the predicate does not hold
  - If the set is empty, there is no such member, so True
  - "All people 12-feet tall will be given free chocolate."
- Existentially quantified expressions are False
  - Definition: there is at least one member of the set for which the predicate holds
- If the set is empty, there is no such member, so False
- Common convention in set theory, arbitrary but settled

Not Just Arrays: Any "Iterable" Objects

Ada 2012

- Those that can be iterated over
- Language-defined, such as the containers
- User-defined too

```
package Characters is new
   Ada.Containers.Vectors (Positive, Character);
use Characters;
Alphabet  : constant Vector := To_Vector('A',1) & 'B' & 'C';
Any_Zed   : constant Boolean :=
           (for some C of Alphabet => C = 'Z');
All_Lower : constant Boolean :=
            (for all C of Alphabet => Is_Lower (C));
```

Conditional / Quantified Expression Usage

Ada 2012

- Use them when a function would be too heavy
- Don't over-use them!

```
if (for some Component of Answers =>
    Component = Ultimate_Answer)
then
```

- Function names enhance readability
  - So put the quantified expression in a function

    ```
    if At_Least_One_Answered (Answers) then
    ```

- Even in pre/postconditions, use functions containing quantified expressions for abstraction

Quiz

Which declaration(s) is(are) legal?

A. `function F (S : String) return Boolean is (for all C of S => C /= ' ')`

B. `function F (S : String) return Boolean is (not for some C of S => C = ' ')`

C. `function F (S : String) return String is (for all C of S => C)`

D.
```
function F (S : String) return String is
  (if (for all C of S => C /= ' ') then "OK" else
"NOK");
```

Quiz

Which declaration(s) is(are) legal?

A. *function F (S : String) return Boolean is (for all C of S => C /= ' ')*

B. `function F (S : String) return Boolean is (not for some C of S => C = ' ')`

C. `function F (S : String) return String is (for all C of S => C)`

D. *function F (S : String) return String is (if (for all C of S => C /= ' ') then "OK" else "NOK");*

B. Parentheses required around the quantified expression

C. Must return a `Boolean`

Quiz

```ada
type T1 is array (1 .. 3) of Integer;
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code is(are) correctly performs equality check on A and B?

A. `function "=" (A : T1; B : T2) return Boolean is (A = T1 (B))`

B.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (for all E1 of A => (for all E2 of B => E1 = E2));
```

C.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (for some E1 of A => (for some E2 of B => A = B));
```

D.
```ada
function "=" (A : T1; B : T2) return Boolean is
  (for all J in A'Range => A (J) = B (J));
```

Quiz

```
type T1 is array (1 .. 3) of Integer;
type T2 is array (1 .. 3) of Integer;
```

Which piece(s) of code is(are) correctly performs equality check on A and B?

A. *`function "=" (A : T1; B : T2) return Boolean is (A = T1 (B))`*

B. `function "=" (A : T1; B : T2) return Boolean is`
```
(for all E1 of A => (for all E2 of B => E1 = E2));
```

C. `function "=" (A : T1; B : T2) return Boolean is`
```
(for some E1 of A => (for some E2 of B => A = B));
```

D. *`function "=" (A : T1; B : T2) return Boolean is`*
```
(for all J in A'Range => A (J) = B (J));
```

B. Counterexample: `A`` andB=(0, 1, 0)' **returns :ada:'False**

C. Counterexample: `A` = (0, 0, 1) and `B` = `(0, 1, 1)` returns `True`

Quiz

```
type Array1_T is array (1 .. 3) of Integer;
type Array2_T is array (1 .. 3) of Array1_T;
A : Array2_T;
```

The above describes an array A whose elements are arrays of three elements. Which expression would one use to determine if at least one of A's elements are sorted?

A.
```
(for some El of A => (for some Idx in 2 .. 3 =>
    El (Idx) >= El (Idx - 1)));
```

B.
```
(for all El of A => for all Idx in 2 .. 3 =>
    El (Idx) >= El (Idx - 1)));
```

C.
```
(for some El of A => (for all Idx in 2 ..  3 =>
    El (Idx) >= El (Idx - 1)));
```

D.
```
(for all El of A => (for some Idx in 2 .. 3 =>
     El (Idx) >= El (Idx - 1)));
```

Quiz

```
type Array1_T is array (1 .. 3) of Integer;
type Array2_T is array (1 .. 3) of Array1_T;
A : Array2_T;
```

The above describes an array A whose elements are arrays of three elements. Which expression would one use to determine if at least one of A's elements are sorted?

A. `(for some El of A => (for some Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));`
B. `(for all El of A => for all Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));`
C. *`(for some El of A => (for all Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));`*
D. `(for all El of A => (for some Idx in 2 .. 3 => El (Idx) >= El (Idx - 1)));`

A. Will be `True` if any element has two consecutive increasing values
B. Will be `True` if every element is sorted
C. Correct
D. Will be `True` if every element has two consecutive increasing values

Lab

Advanced Expressions Lab

- Requirements
  - Allow the user to fill a list with dates
  - After the list is created, use *quantified expressions* to print True/False
    - If any date is not legal (taking into account leap years!)
    - If all dates are in the same calendar year
  - Use *expression functions* for all validation routines
- Hints
  - Use subtype membership for range validation
  - You will need *conditional expressions* in your functions
  - You *can* use component-based iterations for some checks
    - But you *must* use indexed-based iterations for others

Advanced Expressions Lab Solution - Checks

```
subtype Year_T is Positive range 1_900 .. 2_099;
subtype Month_T is Positive range 1 .. 12;
subtype Day_T is Positive range 1 .. 31;

type Date_T is record
   Year  : Positive;
   Month : Positive;
   Day   : Positive;
end record;

List : array (1 .. 5) of Date_T;
Item : Date_T;

function Is_Leap_Year (Year : Positive)
                       return Boolean is
  (Year mod 400 = 0 or else (Year mod 4 = 0 and Year mod 100 /= 0));

function Days_In_Month (Month : Positive;
                        Year  : Positive)
                        return Day_T is
  (case Month is when 4 | 6 | 9 | 11 => 30,
     when 2 => (if Is_Leap_Year (Year) then 29 else 28), when others => 31);

function Is_Valid (Date : Date_T)
                   return Boolean is
  (Date.Year in Year_T and then Date.Month in Month_T
   and then Date.Day <= Days_In_Month (Date.Month, Date.Year));

function Any_Invalid return Boolean is
  (for some Date of List => not Is_Valid (Date));

function Same_Year return Boolean is
  (for all I in List'range => List (I).Year = List (List'first).Year);
```

Advanced Expressions Lab Solution - Main

```
   function Number (Prompt : String)
                    return Positive is
   begin
      Put (Prompt & "> ");
      return Positive'Value (Get_Line);
   end Number;

begin

   for I in List'Range loop
      Item.Year  := Number ("Year");
      Item.Month := Number ("Month");
      Item.Day   := Number ("Day");
      List (I)   := Item;
   end loop;

   Put_Line ("Any invalid: " & Boolean'image (Any_Invalid));
   Put_Line ("Same Year: " & Boolean'image (Same_Year));

end Main;
```

Summary

Summary

- Quantified expressions are general purpose but especially useful with pre/postconditions
  - Consider hiding them behind expressive function names

Limited Types

Introduction

Views

- Specify how values and objects may be manipulated
- Are implicit in much of the language semantics
  - Constants are just variables without any assignment view
  - Task types, protected types implicitly disallow assignment
  - Mode **in** formal parameters disallow assignment

```
Variable : Integer := 0;
...
-- P's view of X prevents modification
procedure P( X :  in  Integer ) is
begin
    ...
end P;
...
P( Variable );
```

Limited Type Views' Semantics

- Prevents copying via predefined assignment
  - Disallows assignment between objects
  - Must make your own **copy** procedure if needed

```
type File is limited ...
...
F1, F2 : File;
...
F1 := F2; -- compile error
```

- Prevents incorrect comparison semantics
  - Disallows predefined equality operator
  - Make your own equality function = if needed

Inappropriate Copying Example

```
type File is ...
F1, F2 : File;
...
Open (F1);
Write ( F1, "Hello" );
-- What is this assignment really trying to do?
F2 := F1;
```

Intended Effects of Copying

```
type File is ...
F1, F2 : File;
...
Open (F1);
Write ( F1, "Hello" );
Copy (Source => F1, Target => F2);
```

Declarations

Examples

```
with Interfaces;
package Multiprocessor_Mutex is
   subtype Id_T is String (1 .. 4);
   -- prevent copying of a lock
   type Limited_T is limited record
      Flag : Interfaces.Unsigned_8;
   end record;
   type Also_Limited_T is record
      Lock : Limited_T;
      Id   : Id_T;
   end record;
   procedure Lock (This : in out Also_Limited_T) is null;
   procedure Unlock (This : in out Also_Limited_T) is null;
end Multiprocessor_Mutex;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/120_limited_types.html#declarations

Limited Type Declarations

- Syntax
  - Additional keyword limited added to record type declaration

```
type defining_identifier is limited record
    component_list
end record;
```

- Are always record types unless also private
  - More in a moment...

Approximate Analog In C++

```
class Stack {
public:
  Stack();
  void Push (int X);
  void Pop (int& X);
  ...
private:
  ...
  // assignment operator hidden
  Stack& operator= (const Stack& other);
}; // Stack
```

Spin Lock Example

```
with Interfaces;
package Multiprocessor_Mutex is
  -- prevent copying of a lock
  type Spin_Lock is limited record
    Flag : Interfaces.Unsigned_8;
  end record;
  procedure Lock  (This : in out Spin_Lock);
  procedure Unlock  (This : in out Spin_Lock);
  pragma Inline (Lock, Unlock);
end Multiprocessor_Mutex;
```

Parameter Passing Mechanism

- Always "by-reference" if explicitly limited
  - Necessary for various reasons (`task` and `protected` types, etc)
  - Advantageous when required for proper behavior
- By definition, these subprograms would be called concurrently
  - Cannot operate on copies of parameters!

```
procedure Lock   (This : in out Spin_Lock);
procedure Unlock (This : in out Spin_Lock);
```

Composites with Limited Types

- Composite containing a limited type becomes limited as well
  - Example: Array of limited elements
    - Array becomes a limited type
  - Prevents assignment and equality loop-holes

```
declare
  -- if we can't copy component S, we can't copy User_Type
  type User_Type is record -- limited because S is limited
    S : File;
    ...
  end record;
  A, B : User_Type;
begin
  A := B;  -- not legal since limited
  ...
end;
```

Quiz

```
type T is limited record
   I : Integer;
end record;

L1, L2 : T;
B : Boolean;
```

Which statement(s) is(are) legal?

A. `L1.I := 1`
B. `L1 := L2`
C. `B := (L1 = L2)`
D. `B := (L1.I = L2.I)`

Quiz

```
type T is limited record
   I : Integer;
end record;

L1, L2 : T;
B : Boolean;
```

Which statement(s) is(are) legal?

A. *`L1.I := 1`*
B. `L1 := L2`
C. `B := (L1 = L2)`
D. *`B := (L1.I = L2.I)`*

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which of the following declaration(s) is(are) legal?

A. `function "+" (A : T) return T is (A)`
B. `function "-" (A : T) return T is (I => -A.I)`
C. `function "=" (A, B : T) return Boolean is (True)`
D. `function "=" (A, B : T) return Boolean is (A.I = T'(I => B.I).I)`

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which of the following declaration(s) is(are) legal?

A. `function "+" (A : T) return T is (A)`
B. *`function "-" (A : T) return T is (I => -A.I)`*
C. *`function "=" (A, B : T) return Boolean is (True)`*
D. *`function "=" (A, B : T) return Boolean is (A.I = T'(I => B.I).I)`*

Quiz

```
package P is
   type T is limited null record;
   type R is record
      F1 : Integer;
      F2 : T;
   end record;
end P;

with P;
procedure Main is
   T1, T2 : P.T;
   R1, R2 : P.R;
begin
```

Which assignment is legal?

A. `T1    := T2;`
B. `R1    := R2;`
C. `R1.F1 := R2.F1;`
D. `R2.F2 := R2.F2;`

Quiz

```
package P is
   type T is limited null record;
   type R is record
      F1 : Integer;
      F2 : T;
   end record;
end P;

with P;
procedure Main is
   T1, T2 : P.T;
   R1, R2 : P.R;
begin
```

Which assignment is legal?

A. `T1    := T2;`
B. `R1    := R2;`
C. *`R1.F1 := R2.F1;`*
D. `R2.F2 := R2.F2;`

Explanations

A. T1 and T2 are `limited` types
B. R1 and R2 contain `limited` types so they are also `limited`
C. Theses components are not `limited` types
D. These components are of a `limited` type

Creating Values

Examples

```
with Interfaces;
package Multiprocessor_Mutex is
   subtype Id_T is String (1 .. 4);
   -- prevent copying of a lock
   type Limited_T is limited record
      Flag : Interfaces.Unsigned_8;
   end record;
   type Also_Limited_T is record
      Lock : Limited_T;
      Id   : Id_T;
   end record;
   procedure Lock (This : in out Also_Limited_T);
   procedure Unlock (This : in out Also_Limited_T);
   function Create (Flag : Interfaces.Unsigned_8;
                    Id   : Id_T)
                    return Also_Limited_T;
end Multiprocessor_Mutex;

package body Multiprocessor_Mutex is
   procedure Lock (This : in out Also_Limited_T) is null;
   procedure Unlock (This : in out Also_Limited_T) is null;
   Global_Lock : Also_Limited_T := (Lock => (Flag => 0), Id => "GLOB");
   function Create (Flag : Interfaces.Unsigned_8;
                    Id   : Id_T)
                    return Also_Limited_T is
      Local_Lock : Also_Limited_T := (Lock => (Flag => 1), Id => "LOCA");
   begin
      Global_Lock.Lock.Flag := Flag;
      Local_Lock.Id         := Id;
      -- Compile error
      -- return Local_Lock;
      -- Compile error
      -- return Global_Lock;
      return (Lock => (Flag => Flag), Id => Id);
   end Create;
end Multiprocessor_Mutex;

with Ada.Text_IO;          use Ada.Text_IO;
with Multiprocessor_Mutex; use Multiprocessor_Mutex;
procedure Perform_Lock is
   Lock1 : Also_Limited_T := (Lock => (Flag => 2), Id => "LOCK");
   Lock2 : Also_Limited_T;
begin
   -- Lock2 := Create ( 3, "CREA" ); -- illegal
   Put_Line (Lock1.Id & Lock1.Lock.Flag'Image);
end Perform_Lock;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/120_limited_types.html#creating-values

Creating Values

- Initialization is not assignment (but looks like it)!
- Via **limited constructor functions**
  - Functions returning values of limited types
- Via **limited aggregates**
  - Aggregates for limited types

```
type Spin_Lock is limited record
  Flag : Interfaces.Unsigned_8;
end record;
...
Mutex : Spin_Lock := (Flag => 0); -- limited aggregate
```

Other Uses for Limited Aggregates

- Values for constant declarations
- Components of enclosing array and record types
- Default expressions for record components
- Expression in an initialized allocator
- Actual parameters for formals of mode `in`
- Results of function return statements
- Defaults for mode `in` formal parameters
- But not right-hand side of assignment statements!

Only Mode **in** for Limited Aggregates

- Aggregates are not variables, so no place to put the returning values for **out** or **in out** formals

```
-- allowed, but not helpful
procedure Wrong_Mode_For_Agg (This : in out Spin_Lock) is
begin
  Lock (This);
  ...
  Unlock (This);
end Wrong_Mode_For_Agg;
...
-- not allowed
Wrong_Mode_For_Agg ( This => (Flag => 0) );
-- allowed
procedure Foo ( Param : access Spin_Lock );
```

Limited Constructor Functions

- Allowed wherever limited aggregates are allowed
- More capable (can perform arbitrary computations)
- Necessary when limited type is also private
  - Users won't have visibility required to express aggregate contents

```
function F return Spin_Lock
is
begin
   ...
  return (Flag => 0);
end F;
```

Writing Limited Constructor Functions

- Remember - copying is not allowed

```
function F return Spin_Lock is
  Local_X : Spin_Lock;
begin
  ...
  return Local_X; -- this is a copy - not legal
   -- (also illegal because of pass-by-reference)
end F;
```

```
Global_X : Spin_Lock;
function F return Spin_Lock is
begin
  ...
  -- This is not legal staring with Ada2005
  return Global_X; -- this is a copy
end F;
```

"Built In-Place"

- Limited aggregates and functions, specifically
- No copying done by implementation
  - Values are constructed in situ

```
Mutex : Spin_Lock := (Flag => 0);
```

```
function F return Spin_Lock is
begin
  return (Flag => 0);
end F;
```

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which piece(s) of code is a legal constructor for T?

A.
```
function F return T is
begin
  return T (I => 0);
end F;
```

B.
```
function F return T is
  Val :  Integer := 0;
begin
  return (I => Val);
end F;
```

C.
```
function F return T is
  Ret : T := (I => 0);
begin
  return Ret;
end F;
```

D.
```
function F return T is
begin
  return (0);
end F;
```

Quiz

```
type T is limited record
   I : Integer;
end record;
```

Which piece(s) of code is a legal constructor for T?

A.
```
function F return T is
begin
  return T (I => 0);
end F;
```

B.
```
function F return T is
  Val :  Integer := 0;
begin
  return (I => Val);
end F;
```

C.
```
function F return T is
  Ret : T := (I => 0);
begin
  return Ret;
end F;
```

D.
```
function F return T is
begin
  return (0);
end F;
```

Quiz

```
package P is
   type T is limited record
      F1 : Integer;
      F2 : Character;
   end record;
   Zero : T := (0, ' ');
   One : constant T := (1, 'a');
   Two : T;
   function F return T;
end P;
```

Which is a correct completion of F?

A. `return (3, 'c');`

B. `Two := (2, 'b');`
   `return Two;`

C. `return One;`

D. `return Zero;`

Quiz

```
package P is
   type T is limited record
      F1 : Integer;
      F2 : Character;
   end record;
   Zero : T := (0, ' ');
   One : constant T := (1, 'a');
   Two : T;
   function F return T;
end P;
```

Which is a correct completion of F?

A. *return (3, 'c');*

B. `Two := (2, 'b');`
   `return Two;`

C. `return One;`

D. `return Zero;`

`A` contains an "in-place" return. The rest all rely on other objects, which would require an (illegal) copy.

Extended Return Statements

Examples

```
with Interfaces; use Interfaces;
package Multiprocessor_Mutex is
   subtype Id_T is String (1 .. 4);
   -- prevent copying of a lock
   type Limited_T is limited record
      Flag : Interfaces.Unsigned_8;
   end record;
   type Also_Limited_T is record
      Lock : Limited_T;
      Id   : Id_T;
   end record;
   procedure Lock (This : in out Also_Limited_T);
   procedure Unlock (This : in out Also_Limited_T);
   function Create (Id : Id_T) return Also_Limited_T;
end Multiprocessor_Mutex;

package body Multiprocessor_Mutex is
   procedure Lock (This : in out Also_Limited_T) is null;
   procedure Unlock (This : in out Also_Limited_T) is null;

   Global_Lock_Counter : Interfaces.Unsigned_8 := 0;
   function Create (Id : Id_T) return Also_Limited_T is
   begin
      return Ret_Val : Also_Limited_T do
         if Global_Lock_Counter = Interfaces.Unsigned_8'Last then
            return;
         end if;
         Global_Lock_Counter := Global_Lock_Counter + 1;
         Ret_Val.Id          := Id;
         Ret_Val.Lock.Flag   := Global_Lock_Counter;
      end return;
   end Create;
end Multiprocessor_Mutex;

with Ada.Text_IO;          use Ada.Text_IO;
with Multiprocessor_Mutex; use Multiprocessor_Mutex;
procedure Perform_Lock is
   Lock1 : constant Also_Limited_T := Create ("One ");
   Lock2 : constant Also_Limited_T := Create ("Two ");
begin
   Put_Line (Lock1.Id & Lock1.Lock.Flag'Image);
   Put_Line (Lock2.Id & Lock2.Lock.Flag'Image);
end Perform_Lock;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/120_limited_types.html#extended-return-statements

Function Extended Return Statements

Ada 2005

- *Extended return*
- Result is expressed as an object
- More expressive than aggregates
- Handling of unconstrained types
- Syntax (simplified):

```
return identifier : subtype [:= expression];

return identifier : subtype
[do
   sequence_of_statements ...
 end return];
```

Extended Return Statements Example

```
--  Implicitely limited array
type Spin_Lock_Array (Positive range <>) of Spin_Lock;

function F return Spin_Lock_Array is
begin
  return Result : Spin_Lock_Array (1 .. 10) do
    ...
  end return;
end F;
```

Expression / Statements Are Optional

Ada 2005

- Without sequence (returns default if any)

```
function F return Spin_Lock is
begin
  return Result : Spin_Lock;
end F;
```

- With sequence

```
function F return Spin_Lock is
  X : Interfaces.Unsigned_8;
begin
  --  compute X ...
  return Result : Spin_Lock := (Flag => X);
end F;
```

Statements Restrictions

Ada 2005

- **No** nested extended return
- **Simple** return statement **allowed**
  - **Without** expression
  - Returns the value of the **declared object** immediately

```
function F return Spin_Lock is
begin
  return Result : Spin_Lock do
    if Set_Flag then
      Result.Flag := 1;
      return;  --  returns 'Result'
    end if;
    Result.Flag := 0;
  end return; --  Implicit return
end F;
```

Quiz

```
type T is limited record
   I : Integer;
end record;
O : T := F;
```

Which declaration(s) of `F` is(are) valid?

A. `function F return T is (I := 1)`

B. `function F return T is (I => 1)`

C. `function F return T is (1)`

D.
```
function F return T is
begin
  return R : T do
    R.I := 1;
  end return;
end F;
```

Quiz

```ada
type T is limited record
   I : Integer;
end record;
O : T := F;
```

Which declaration(s) of `F` is(are) valid?

A. `function F return T is (I := 1)`

B. *`function F return T is (I => 1)`*

C. `function F return T is (1)`

D. *`function F return T is`*
```ada
begin
  return R : T do
    R.I := 1;
  end return;
end F;
```

Combining Limited and Private Views

Examples

```
with Interfaces; use Interfaces;
package Multiprocessor_Mutex is
   type Limited_T is limited private;
   procedure Lock (This : in out Limited_T);
   procedure Unlock (This : in out Limited_T);
   function Create return Limited_T;
private
   type Limited_T is limited -- no internal copying allowed
   record
      Flag : Interfaces.Unsigned_8; -- users cannot see this
   end record;
end Multiprocessor_Mutex;

package body Multiprocessor_Mutex is
   procedure Lock (This : in out Limited_T) is null;
   procedure Unlock (This : in out Limited_T) is null;

   Global_Lock_Counter : Interfaces.Unsigned_8 := 0;
   function Create return Limited_T is
   begin
      return Ret_Val : Limited_T do
         Global_Lock_Counter := Global_Lock_Counter + 1;
         Ret_Val.Flag        := Global_Lock_Counter;
      end return;
   end Create;
end Multiprocessor_Mutex;

with Multiprocessor_Mutex; use Multiprocessor_Mutex;
package Use_Limited_Type is
   type Legal is limited private;
   type Also_Legal is limited private;
   -- type Not_Legal is private;
   -- type Also_Not_Legal is private;
private
   type Legal is record
      S : Limited_T;
   end record;
   type Also_Legal is limited record
      S : Limited_T;
   end record;
   -- type Not_Legal is limited record
   --    S : Limited_T;
   -- end record;
   -- type Also_Not_Legal is record
   --    S : Limited_T;
   -- end record;
end Use_Limited_Type;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/120_limited_types.html#combining-limited-and-private-views

Limited Private Types

- A combination of **limited** and **private** views
  - No client compile-time visibility to representation
  - No client assignment or predefined equality
- The typical design idiom for **limited** types
- Syntax
  - Additional reserved word **limited** added to **private** type declaration

```
type defining_identifier is limited private;
```

Limited Private Type Rationale (1)

```
package Multiprocessor_Mutex is
  -- copying is prevented
  type Spin_Lock is limited record
    -- but users can see this!
    Flag : Interfaces.Unsigned_8;
  end record;
  procedure Lock (This : in out Spin_Lock);
  procedure Unlock (This : in out Spin_Lock);
  pragma Inline (Lock, Unlock);
end Multiprocessor_Mutex;
```

Limited Private Type Rationale (2)

```ada
package MultiProcessor_Mutex is
  -- copying is prevented AND users cannot see contents
  type Spin_Lock is limited private;
  procedure Lock (The_Lock : in out Spin_Lock);
  procedure Unlock (The_Lock : in out Spin_Lock);
  pragma Inline (Lock, Unlock);
private
  type Spin_Lock is ...
end MultiProcessor_Mutex;
```

Limited Private Type Completions

- Clients have the partial view as **limited** and **private**
- The full view completion can be any kind of type
- Not required to be a record type just because the partial view is limited

```
package P is
  type Unique_ID_T is limited private;
  ...
private
  type Unique_ID_T is range 1 .. 10;
end P;
```

Write-Only Register Example

```
package Write_Only is
  type Byte is limited private;
  type Word is limited private;
  type Longword is limited private;
  procedure Assign (Input : in Unsigned_8;
                    To    : in out Byte);
  procedure Assign (Input : in Unsigned_16;
                    To    : in out Word);
  procedure Assign (Input : in Unsigned_32;
                    To    : in out Longword);
private
  type Byte is new Unsigned_8;
  type Word is new Unsigned_16;
  type Longword is new Unsigned_32;
end Write_Only;
```

Explicitly Limited Completions

- Completion in Full view includes word **limited**
- Optional
- Requires a record type as the completion

```
package MultiProcessor_Mutex is
  type Spin_Lock is limited private;
  procedure Lock (This : in out Spin_Lock);
  procedure Unlock (This : in out Spin_Lock);
private
  type Spin_Lock is limited -- full view is limited as well
    record
      Flag : Interfaces.Unsigned_8;
    end record;
end MultiProcessor_Mutex;
```

Effects of Explicitly Limited Completions

- Allows no internal copying too
- Forces parameters to be passed by-reference

```
package MultiProcessor_Mutex is
  type Spin_Lock is limited private;
  procedure Lock (This : in out Spin_Lock);
  procedure Unlock (This : in out Spin_Lock);
private
  type Spin_Lock is limited record
    Flag : Interfaces.Unsigned_8;
  end record;
end MultiProcessor_Mutex;
```

Automatically Limited Full View

- When other limited types are used in the representation
- Recall composite types containing limited types are **limited** too

```
package Foo is
   type Legal is limited private;
   type Also_Legal is limited private;
   type Not_Legal is private;
   type Also_Not_Legal is private;
private
   type Legal is record
      S : A_Limited_Type;
   end record;
   type Also_Legal is limited record
      S : A_Limited_Type;
   end record;
   type Not_Legal is limited record
      S : A_Limited_Type;
   end record;
   type Also_Not_Legal is record
      S : A_Limited_Type;
   end record;
end Foo;
```

Quiz

```
package P is
   type Priv is private;
private
   type Lim is limited null record;
end P;
```

Which of the following piece(s) of code is(are) legal?

A.
```
type Priv is record
  E : Lim;
end record;
```

B.
```
type Priv is record
  E : Float;
end record;
```

C.
```
type A is array (1 .. 10) of Lim;
type Priv is record
  F : A;
end record;
```

D.
```
type Acc is access Lim;
type Priv is record
  F : Acc;
end record;
```

Quiz

```
package P is
   type Priv is private;
private
   type Lim is limited null record;
end P;
```

Which of the following piece(s) of code is(are) legal?

A.
```
type Priv is record
   E : Lim;
end record;
```

B.
```
type Priv is record
   E : Float;
end record;
```

C.
```
type A is array (1 .. 10) of Lim;
type Priv is record
   F : A;
end record;
```

D.
```
type Acc is access Lim;
type Priv is record
   F : Acc;
end record;
```

A. E has limited type, partial view of Priv must be `private limited`

Quiz

```
package P is
   type L1_T is limited private;
   type L2_T is limited private;
   type P1_T is private;
   type P2_T is private;
private
   type L1_T is limited record
      Field : Integer;
   end record;
   type L2_T is record
      Field : Integer;
   end record;
   type P1_T is limited record
      Field : L1_T;
   end record;
   type P2_T is record
      Field : L2_T;
   end record;
```

What will happen when the above code is compiled?

A. Type P1_T will generate a compile error
B. Type P2_T will generate a compile error
C. Both type P1_T and type P2_T will generate compile errors
D. The code will compile successfully

Quiz

```
package P is
   type L1_T is limited private;
   type L2_T is limited private;
   type P1_T is private;
   type P2_T is private;
private
   type L1_T is limited record
      Field : Integer;
   end record;
   type L2_T is record
      Field : Integer;
   end record;
   type P1_T is limited record
      Field : L1_T;
   end record;
   type P2_T is record
      Field : L2_T;
   end record;
```

What will happen when the above code is compiled?

A. ***Type P1_T will generate a compile error***
B. Type P2_T will generate a compile error
C. Both type P1_T and type P2_T will generate compile errors
D. The code will compile successfully

The full definition of type `P1_T` adds additional restrictions, which is not allowed. Although `P2_T` contains a component whose visible view is `limited`, the internal view is not `limited` so `P2_T` is not `limited`.

Lab

Limited Types Lab

- Requirements
  - Create an employee record data type consisting of a name, ID, hourly pay rate
    - ID should be unique for every record
  - Create a timecard record data type consisting of an employee record, hours worked, and total pay
  - Create a main program that generates timecards and prints their contents
- Hints
  - If the ID is unique, that means we cannot copy employee records

Limited Types Lab Solution - Employee Data (Spec)

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Employee_Data is

  type Employee_T is limited private;
  type Hourly_Rate_T is delta 0.01 digits 6 range 0.0 .. 999.99;
  type Id_T is range 999 .. 9_999;

  function Create (Name : String;
                   Rate : Hourly_Rate_T := 0.0)
                   return Employee_T;
  function Id (Employee : Employee_T) return Id_T;
  function Name (Employee : Employee_T) return String;
  function Rate (Employee : Employee_T) return Hourly_Rate_T;

private
  type Employee_T is limited record
    Name : Unbounded_String := Null_Unbounded_String;
    Rate : Hourly_Rate_T    := 0.0;
    Id   : Id_T             := Id_T'First;
  end record;
end Employee_Data;
```

Limited Types Lab Solution - Timecards (Spec)

```
with Employee_Data;
package Timecards is

  type Hours_Worked_T is digits 3 range 0.0 .. 24.0;
  type Pay_T is digits 6;
  type Timecard_T is limited private;

  function Create (Name  : String;
                   Rate  : Employee_Data.Hourly_Rate_T;
                   Hours : Hours_Worked_T)
                   return Timecard_T;

  function Id (Timecard : Timecard_T) return Employee_Data.Id_T;
  function Name (Timecard : Timecard_T) return String;
  function Rate (Timecard : Timecard_T) return Employee_Data.Hourly_Rate_T;
  function Pay (Timecard : Timecard_T) return Pay_T;
  function Image ( Timecard : Timecard_T ) return String;

private
  type Timecard_T is limited record
    Employee     : Employee_Data.Employee_T;
    Hours_Worked : Hours_Worked_T := 0.0;
    Pay          : Pay_T          := 0.0;
  end record;
end Timecards;
```

Limited Types Lab Solution - Employee Data (Body)

```
package body Employee_Data is

  Last_Used_Id : Id_T := Id_T'First;

  function Create (Name : String;
                   Rate : Hourly_Rate_T := 0.0)
                   return Employee_T is
  begin
    return Ret_Val : Employee_T do
      Last_Used_Id := Id_T'Succ (Last_Used_Id);
      Ret_Val.Name := To_Unbounded_String (Name);
      Ret_Val.Rate := Rate;
      Ret_Val.Id   := Last_Used_Id;
    end return;
  end Create;

  function Id (Employee : Employee_T) return Id_T is (Employee.Id);
  function Name (Employee : Employee_T) return String is (To_String (Employee.Name));
  function Rate (Employee : Employee_T) return Hourly_Rate_T is (Employee.Rate);

end Employee_Data;
```

Limited Types Lab Solution - Timecards (Body)

```
package body Timecards is

  function Create (Name  : String;
                   Rate  : Employee_Data.Hourly_Rate_T;
                   Hours : Hours_Worked_T)
                   return Timecard_T is
  begin
    return (Employee     => Employee_Data.Create (Name, Rate),
            Hours_Worked => Hours,
            Pay          => Pay_T (Hours) * Pay_T (Rate));
  end Create;

  function Id (Timecard : Timecard_T) return Employee_Data.Id_T is
     (Employee_Data.Id (Timecard.Employee));
  function Name (Timecard : Timecard_T) return String is
     (Employee_Data.Name (Timecard.Employee));
  function Rate (Timecard : Timecard_T) return Employee_Data.Hourly_Rate_T is
     (Employee_Data.Rate (Timecard.Employee));
  function Pay (Timecard : Timecard_T) return Pay_T is
     (Timecard.Pay);

  function Image (Timecard : Timecard_T) return String is
    Name_S : constant String := Name (Timecard);
    Id_S   : constant String := Employee_Data.Id_T'Image (Employee_Data.Id (Timecard.Employee));
    Rate_S : constant String := Employee_Data.Hourly_Rate_T'Image
                                 (Employee_Data.Rate (Timecard.Employee));
    Hours_S : constant String := Hours_Worked_T'Image (Timecard.Hours_Worked);
    Pay_S   : constant String := Pay_T'Image (Timecard.Pay);
  begin
    return Name_S & " ( " & Id_S & " ) => " & Hours_S & " hours * " & Rate_S & "/hour = " & Pay_S;
  end Image;
end Timecards;
```

Limited Types Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Timecards;
procedure Main is

   One : constant Timecards.Timecard_T :=
         Timecards.Create (Name  => "Fred Flintstone",
                           Rate  => 1.1,
                           Hours => 2.2);
   Two : constant Timecards.Timecard_T :=
         Timecards.Create (Name  => "Barney Rubble",
                           Rate  => 3.3,
                           Hours => 4.4);

begin
   Put_Line ( Timecards.Image ( One ) );
   Put_Line ( timecards.Image ( Two ) );
end Main;
```

Summary

Summary

- Limited view protects against improper operations
  - Incorrect equality semantics
  - Copying via assignment
- Enclosing composite types are `limited` too
  - Even if they don't use keyword `limited` themselves
- Limited types are always passed by-reference
- Extended return statements work for any type
  - Ada 2005 and later
- Don't make types `limited` unless necessary
  - Users generally expect assignment to be available

Advanced Privacy

Type Views

Capabilities / Constraints Of A Type

- *Constraints* in a type declaration
  - Reduce the set of operations available on a type
  - `limited`
  - Discriminants
  - `abstract`
- *Capabilities* in a type declaration
  - Extends or modifies the set of operations available on a type
  - `tagged`
  - Tagged extensions

Partial Vs Full View Of A Type

- If the partial view declares **capabilities**, the full view **must provide** them
  - Full view may provide supplementary capabilities undeclared in the partial view
- If the full has **constraints**, the partial view **must declare** them
  - Partial view may declare supplementary constraint that the full view doesn't have

```
package P is
   type T is limited private;
   -- Does not need to declare any capability
   -- Declares a constraint: limited
private
   type T is tagged null record;
   -- Declares a capability: tagged
   -- Does not need to declare any constraint
end P;
```

Discriminants

- Discriminants with no default must be declared both on the partial and full view

```
package P is
   type T (V : Integer) is private;
private
   type T (V : Integer) is null record;
end P;
```

- Discriminants with default (in the full view) may be omitted by the partial view

```
package P is
   type T1 (V : Integer := 0) is private;
   type T2 is private;
private
   type T1 (V : Integer := 0) is null record;
   type T2 (V : Integer := 0) is null record;
end P;
```

Unknown Constraint

- It is possible to establish that the type is unconstrained without any more information
- Constrained and unconstrained types can complete the private declaration

```
package P is
   type T1 (<>) is private;
   type T2 (<>) is private;
   type T3 (<>) is private;
private
   type T1 (V : Integer) is null record;
   type T2 is array (Integer range <>) of Integer;
   type T3 is range 1 .. 10;
end P;
```

Limited

- Limited property can apply only to the partial view
- If the full view is implicitly limited, the partial view has to be explicitly limited

```
package P is
   type T1 is limited private;
   type T2 is limited private;
   type T3 is limited private;
private
   type T1 is limited null record;
   type T2 is record
      V : T1;
   end record;
   type T3 is range 1 .. 10;
end P;
```

Tagged

- If the partial view is tagged, the full view has to be tagged
- The partial view can hide the fact that the type is tagged in the full view

```
package P is
   type T1 is private;
   type T2 is tagged private;
   type T3 is tagged private;
private
   type T1 is tagged null record;
   type T2 is tagged null record;
   type T3 is new T2 with null record;
end P;
```

- Primitives can be either public or private
  - Except when they **have** to be derived (constructor functions or abstract subprograms)

Tagged Extension

- The partial view may declare an extension
- The actual extension can be done on the same type, or on any of its children

```
package P is
   type Root        is tagged private;
   type Child       is new Root with private;
   type Grand_Child is new Root with private;
private
   type Root        is tagged null record;
   type Child       is new Root with null record;
   type Grand_Child is new Child with null record;
end P;
```

Tagged Abstract

- Partial view may be abstract even if Full view is not
- If Full view is abstract, Private view has to be so

```
package P is
   type T1 is abstract tagged private;
   type T2 is abstract tagged private;
private
   type T1 is abstract tagged null record;
   type T2 is tagged null record;
end P;
```

- Abstract primitives have to be public (otherwise, clients couldn't derive)

Protection Idiom

- It is possible to declare an object that can't be copied, and has to be initialized through a constructor function

```
package P is
   type T (<>) is limited private;
   function F return T;
private
   type T is null record;
end P;
```

- Helps keeping track of the object usage

Quiz

```
type T is private;
```

Which completion(s) is(are) correct for the type T?

A. `type T is tagged null record`
B. `type T is limited null record`
C. `type T is array (1 .. 10) of Integer`
D. `type T is abstract tagged null record`

Quiz

```
type T is private;
```

Which completion(s) is(are) correct for the type `T`?

A. *`type T is tagged null record`*
B. `type T is limited null record`
C. *`type T is array (1 .. 10) of Integer`*
D. `type T is abstract tagged null record`

Incomplete Types

Incomplete Types

- An *incomplete type* is a premature view on a type
    - Does specify the type name
    - Can specify the type discriminants
    - Can specify if the type is tagged
- It can be used in contexts where minimum representation information is required
    - In declaration of access types
    - In subprograms specifications (only if the body has full visibility on the representation)
    - As formal parameter of generics accepting an incomplete type

How To Get An Incomplete Type View?

- From an explicit declaration

```
type T;
type T_Access is access all T;
type T is record
   V : T_Access;
end record;
```

- From a **limited with** (see section on packages)
- From an incomplete generic formal parameter (see section on generics)

```
generic
   type T;
   procedure Proc (V:T);
package P is
   ...
end P;
```

Type Completion Deferred To The Body

- In the private part of a package, it is possible to defer the completion of an incomplete type to the body
- This allows to completely hide the implementation of a type

```
package P is
   ...
private
   type T;
   procedure P (V : T);
   X : access T;
end P;
package body P is
   type T is record
      A, B : Integer;
   end record;
   ...
end P;
```

Quiz

```
type T;
```

Which of the following types is(are) legal?

A. `type Acc is access T`

B. `type Arr is array (1 .. 10) of T`

C. `type T2 is new T`

D.
```
type T2 is record
  Acc : access T;
end record;
```

Quiz

```
type T;
```

Which of the following types is(are) legal?

A. *`type Acc is access T`*

B. `type Arr is array (1 .. 10) of T`

C. `type T2 is new T`

D. *type T2 is record*
```
  Acc : access T;
end record;
```

D. Be careful about the use of an anonymous type here!

Quiz

```
package Pkg is
   type T is private;
private
```

Which of the following completion(s) of `T` is(are) valid?

A. `type T`
B. `type T is tagged null record`
C. `type T is limited null record`
D. `type T is new Integer`

Quiz

```
package Pkg is
   type T is private;
private
```

Which of the following completion(s) of `T` is(are) valid?

A. `type T`
B. *`type T is tagged null record`*
C. `type T is limited null record`
D. *`type T is new Integer`*

Private Library Units

Child Units And Privacy

- Normally, a child public part cannot access a parent private part

```
package Root is
private
   type T is range 1..10;
end Root;
```

```
package Ro
  X1 : T;
private
  X2 : T;
end Root.C
```

- *Private child* can
  - Used for "implementation details"

Importing a Private Child

- A private child can access its parent private part
- Access to a private child is limited
  - *Private descendents of their parent*
  - Parent - visible from body
  - Public siblings - visible from private section, and body
  - Private siblings - visible from public and private sections, and body

```
package Root is
private
  type T is range 1..10;
end Root;
```

```
private package Root.Chi
  X1 : T;
private
  X2 : T;
end Root.Child;

with Root.Child; -- ille
procedure Main is
begin
   Root.Child.X1 := 10;
end Main;
```

Private Children And Dependency

```
private package Root.Child1 is
   type T is range 1 .. 10;
end Root.Child1;
```

- Private package cannot be withed by a public package

```
with Root.Child1; -- illegal
package Root.Child2 is
   X1 : Root.Child1.T; -- illegal
Private
   X2 : Root.Child1.T; -- illegal
end Root.Child2;
```

- They can by a private child or a child body

```
with Root.Child1;
Private package Root.Child2 is
   X1 : Root.Child1.T;
Private
   X2 : Root.Child1.T;
end Root.Child2;
```

- They can be private-withed

```
Private with Root.Child1;
package Root.Child2 is
   X1 : Root.Child1.T; -- illegal
Private
   X2 : Root.Child1.T;
end Root.Child2;
```

- Once something is private, it can never exit the private area

Children "Inherit" From Private Properties Of Parent

- Private property always refers to the direct parent
- Public children of private packages stay private to the outside world
- Private children of private packages restrain even more the accessibility

```
package Root is
end Root;

private package Root.Child is
  -- with allowed on Root body
  -- with allowed on Root children
  -- with forbidden outside of Root
end Root.Child;

package Root.Child.Grand1 is
  -- with allowed on Root body
  -- with allowed on Root children
  -- with forbidden outside of Root
end Root.Child.Grand1;

private package Root.Child.Grand2 is
  -- with allowed on Root.Child body
  -- with allowed on Root.Child children
  -- with forbidden outside of Root.Child
  -- with forbidden on Root
  -- with forbidden on Root children
end Root.Child1.Grand2;
```

Lab

Advanced Privacy Lab

- Requirements
  - Create a package defining a message type whose implementation is solely in the body
    - You will need accessor functions to set / get the content
    - Create a function to return a string representation of the message contents
  - Create another package that defines the types needed for a linked list of messages
    - Each message in the list should have an identifier not visible to any clients
  - Create a package containing simple operations on the list
    - Typical operations like list creation and list traversal
    - Create a subprogram to print the list contents
  - Have your main program add items to the list and then print the list
- Hints
  - You will need to employ some (but not necessarily all) of the techniques discussed in this module

Advanced Privacy Lab Solution - Message Type

```
package Messages is
   type Message_T is private;

   procedure Set_Content (Message : in out Message_T;
                          Value   :        Integer);
   function Content (Message : Message_T) return Integer;
   function Image (Message : Message_T) return String;

private
   type Message_Content_T;
   type Message_T is access Message_Content_T;
end Messages;

package body Messages is
   type Message_Content_T is new Integer;

   procedure Set_Content (Message : in out Message_T;
                          Value   :        Integer) is
      New_Value : constant Message_Content_T := Message_Content_T (Value);
   begin
      if Message = null then
         Message := new Message_Content_T'(New_Value);
      else
         Message.all := New_Value;
      end if;
   end Set_Content;

   function Content (Message : Message_T) return Integer is
      (Integer (Message.all));
   function Image (Message : Message_T) return String is
      ("**" & Message_Content_T'Image (Message.all));
end Messages;
```

Advanced Privacy Lab Solution - Message List Type

```
package Messages.List_Types is
   type List_T is private;
private
   type List_Content_T;
   type List_T is access List_Content_T;
   type Id_Type is range 1_000 .. 9_999;
   type List_Content_T is record
      Id      : Id_Type;
      Content : Message_T;
      Next    : List_T;
   end record;
end Messages.List_Types;
```

Advanced Privacy Lab Solution - Message List Operations

```
package Messages.List_Types.Operations is
   procedure Append (List : in out List_T;
                     Item :        Message_T);
   function Next (List : List_T) return List_T;
   function Is_Null (List : List_T) return Boolean;
   function Image (Message : List_T) return String;
end Messages.List_Types.Operations;

package body Messages.List_Types.Operations is
   Id : Id_Type := Id_Type'First;

   procedure Append (List : in out List_T;
                     Item :        Message_T) is
   begin
      if List = null then
         List := new List_Content_T'(Id => Id, Content => Item, Next => null);
      else
         List.Next := new List_Content_T'(Id => Id, Content => Item, Next => null);
      end if;
      Id := Id_Type'Succ (Id);
   end Append;

   function Next (List : List_T) return List_T is (List.Next);
   function Is_Null (List : List_T) return Boolean is (List = null);

   function Image (Message : List_T) return String is
   begin
      if Is_Null (Message) then
         return "" & ASCII.LF;
      else
         return "id: " & Id_Type'Image (Message.Id) & " => " &
           Image (Message.Content) & ASCII.LF & Image (Message.Next);
      end if;
   end Image;
end Messages.List_Types.Operations;
```

Advanced Privacy Lab Solution - Main

```
with Ada.Text_IO;
with Messages;
with Messages.List_Types;
with Messages.List_Types.Operations;
procedure Main is
   package Types renames Messages.List_Types;
   package Operations renames Messages.List_Types.Operations;

   List : Types.List_T;
   Head : Types.List_T;

   function Convert (Value : Integer) return Messages.Message_T is
      Ret_Value : Messages.Message_T;
   begin
      Messages.Set_Content (Ret_Value, Value);
      return Ret_Value;
   end Convert;

   procedure Add_One (Value : Integer) is
   begin
      Operations.Append (List, Convert (Value));
      List := Operations.Next (List);
   end Add_One;

begin
   Operations.Append (List, Convert (1));
   Head := List;
   Add_One (23);
   Add_One (456);
   Add_One (78);
   Add_One (9);
   Ada.Text_IO.Put_Line (Operations.Image (Head));
end Main;
```

Summary

Summary

- Ada has many mechanisms for data hiding / control
- Start by fully understanding supplier / client relationship
- Need to balance simplicity of interfaces with complexity of structure
  - Small number of relationship per package with many packages
  - Fewer packages with more relationships in each package
  - No set standard
    - Varies from project to project
    - Can even vary within a code base

Advanced Access Types

Introduction

Access Types Design

- Memory addresses objects are called *access types*
- Objects are associated to **pools** of memory
  - With different allocation / deallocation policies

```
type Integer_Pool_Access is access Integer;
P_A : Integer_Pool_Access := new Integer;

type Integer_General_Access is access all Integer;
G : aliased Integer;
G_A1 : Integer_General_Access := G'Access;
G_A2 : Integer_General_Access := new Integer;
```

- This module will only deal with *general access types*

Access Types Can Be Dangerous

- Multiple memory issues
  - Leaks / corruptions
- Introduces potential random failures complicated to analyze
- Increase the complexity of the data structures
- May decrease the performances of the application
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot more expensive than stacking objects
- Ada avoids using accesses as much as possible
  - Arrays are not pointers
  - Many parameters are implicitly passed by reference
- Only use them when needed

Access Types

Declaration Location

- Can be at library level

```
package P is
  type String_Access is access all String;
end P;
```

- Can be nested in a procedure

```
package body P is
   procedure Proc is
      type String_Access is access all String;
   begin
      ...
   end Proc;
end P;
```

- Nesting adds non-trivial issues
  - Creates a nested pool with a nested accessibility
  - Don't do that unless you know what you are doing! (see later)

Access Types and Primitives

- Subprograms using an access type are primitive of the **access type**
    - **Not** the type of the accessed object

```
type A_T is access all T;
procedure Proc (V : A_T); -- Primitive of A_T, not T
```

- Primitive of the type can be created with the `access` mode
    - **Anonymous** access type

```
procedure Proc (V : access T); -- Primitive of T
```

Anonymous Access Types

- Can be declared in several places
    - Must be **pool-specific**
- Make sense as parameters of a primitive
- Else, raises a fundamental issue
    - Two different `access T` are **not** compatible

```
procedure Main is
   A : access Integer;
begin
   declare
      type R is record
         A : access Integer;
      end record;

      D : R := (A => new Integer);
   begin
      -- Invalid, and no conversion possible
      A := D.A;
   end;
end Main;
```

Pool-Specific Access Types

Examples

```
package Pool_Specific is
   type Pointed_To_T is new Integer;
   type Access_T is access Pointed_To_T;
   Object : Access_T := new Pointed_To_T;

   type Other_Access_T is access Pointed_To_T;
   -- Other_Object : Other_Access_T := Other_Access_T ( Object ); -- illegal

   type String_Access_T is access String;
end Pool_Specific;

with Ada.Unchecked_Deallocation;
with Ada.Text_IO;   use Ada.Text_IO;
with Pool_Specific; use Pool_Specific;
procedure Use_Pool_Specific is
   X : Access_T        := new Pointed_To_T'(123);
   Y : String_Access_T := new String (1 .. 10);

   procedure Free is new Ada.Unchecked_Deallocation (Pointed_To_T, Access_T);

begin
   Put_Line (Y.all);
   Y := new String'("String will be long enough to hold this");
   Put_Line (Y.all);
   Put_Line (Pointed_To_T'Image (X.all));
   Free (X);
   Put_Line (Pointed_To_T'Image (X.all)); -- run-time error
end Use_Pool_Specific;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/140_access_types.html#pool-specific-access-types

Pool-Specific Access Type

- An access type is a type

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

- Conversion is **not** possible between pool-specific access types

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 pointers
- 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

Deallocation Example

```
-- generic used to deallocate memory
with Ada.Unchecked_Deallocation;
procedure P is
   type An_Access is access A_Type;
   -- create instances of deallocation function
   -- (object type, access type)
   procedure Free is new Ada.Unchecked_Deallocation
     (A_Type, An_Access);
   V : An_Access := new A_Type;
begin
   Free (V);
   -- V is now null
end P;
```

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
```

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**

Accessibility Checks

Introduction to Accessibility Checks (1/2)

- The depth of an object depends on its nesting within declarative scopes

```
package body P is
   --  Library level, depth 0
   procedure Proc is
      --  Library level subprogram, depth 1
      procedure Nested is
         -- Nested subprogram, enclosing + 1, here 2
      begin
          null;
      end Nested;
   begin
      null;
   end Proc;
end P;
```

- Access types can access objects of the **same or lower** depth
- The compiler checks it statically
    - Removing checks is a workaround!

Introduction to Accessibility Checks (2/2)

- Issues with nesting

```
package body P is
   type T0 is access all Integer;
   A0 : T0;
   V0 : aliased Integer;

   procedure Proc is
      type T1 is access all Integer;
      A1 : T1;
      V1 : aliased Integer;
   begin
      A0 := V0'Access;
      -- A0 := V1'Access; -- illegal
      A0 := V1'Unchecked_Access;
      A1 := V0'Access;
      A1 := V1'Access;
      A1 := T1 (A0);
      A1 := new Integer;
      -- A0 := T0 (A1); -- illegal
  end Proc;
end P;
```

- Simple workaround is to avoid nested access types

Dynamic Accessibility Checks

- Following the same rules
  - Performed dynamically by the runtime
- Lots of possible cases
  - New compiler versions may detect more cases
  - Using access always requires proper debugging and reviewing

```
procedure Main is
   type Acc is access all Integer;
   O : Acc;

   procedure Set_Value (V : access Integer) is
   begin
      O := Acc (V);
   end Set_Value;
begin
   declare
      O2 : aliased Integer := 2;
   begin
      Set_Value (O2'Access);
   end;
end Main;
```

Getting Around Accessibility Checks

- Sometimes it is OK to use unsafe accesses to data
- `'Unchecked_Access` allows access to a variable of an incompatible accessibility level
- Beware of potential problems!

```
type Acc is access all Integer;
G : Acc;
procedure P is
   V : aliased Integer;
begin
   G := V'Unchecked_Access;
   ...
   Do_Something ( G.all ); -- This is "reasonable"
end P;
```

Using Pointers For Recursive Structures

- It is not possible to declare recursive structure
- But there can be an access to the enclosing type

```
type Cell; -- partial declaration
type Cell_Access is access all Cell;
type Cell is record -- full declaration
   Next       : Cell_Access;
   Some_Value : Integer;
end record;
```

Memory Corruption

Common Memory Problems (1/3)

- Uninitialized pointers

```
declare
   type An_Access is access all Integer;
   V : An_Access;
begin
   V.all := 5; -- constraint error
```

- Double deallocation

```
declare
   type An_Access is access all Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   V1 : An_Access := new Integer;
   V2 : An_Access := V1;
begin
   Free (V1);
   ...
   Free (V2);
```

  - May raise `Storage_Error` if memory is still protected (unallocated)
  - May deallocate a different object if memory has been reallocated
    - Putting that object in an inconsistent state

Common Memory Problems (2/3)

- Accessing deallocated memory

```
declare
   type An_Access is access all Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   V1 : An_Access := new Integer;
   V2 : An_Access := V1;
begin
   Free (V1);
   ...
   V2.all := 5;
```

  - May raise `Storage_Error` if memory is still protected (unallocated)
  - May modify a different object if memory has been reallocated (putting that object in an inconsistent state)

Common Memory Problems (3/3)

- Memory leaks

```
declare
   type An_Access is access all Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation (Integer, An_Access);
   V : An_Access := new Integer;
begin
   V := null;
```

  - Silent problem
    - Might raise `Storage_Error` if too many leaks
    - Might slow down the program if too many page faults

How To Fix Memory Problems?

- There is no language-defined solution
- Use the debugger!
- Use additional tools
  - `gnatmem` monitor memory leaks
  - `valgrind` monitor all the dynamic memory
  - **GNAT.Debug_Pools** gives a pool for an access type, raising explicit exception in case of invalid access
  - Others...

Memory Management

Simple Linked List

- A linked list object typically consists of:
  - Content
  - "Indication" of next item in list
    - Fancier linked lists may reference previous item in list
- "Indication" is just a pointer to another linked list object
  - Therefore, self-referencing
- Ada does not allow a record to self-reference

Incomplete Types

- In Ada, an *incomplete type* is just the word **type** followed by the type name
  - Optionally, the name may be followed by (<>) to indicate the full type may be unconstrained
- Ada allows access types to point to an incomplete type
  - Just about the only thing you *can* do with an incomplete type!

```
type Some_Record_T;
type Some_Record_Access_T is access all Some_Record_T;

type Unconstrained_Record_T (<>);
type Unconstrained_Record_Access_T is access all Unconstrained_Record_T;

type Some_Record_T is record
   Field : String (1 .. 10);
end record;

type Unconstrained_Record_T (Size : Index_T) is record
   Field : String (1 .. Size);
end record;
```

Linked List in Ada

- Now that we have a pointer to the record type (by name), we can use it in the full definition of the record type

```
type Some_Record_T is record
   Field : String (1 .. 10);
   Next  : Some_Record_Access_T;
end record;

type Unconstrained_Record_T (Size : Index_T) is record
   Field    : String (1 .. Size);
   Next     : Unconstrained_Record_Access_T;
   Previous : Unconstrained_Record_Access_T;
end record;
```

Simplistic Linked List

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;
procedure Simple is
   type Some_Record_T;
   type Some_Record_Access_T is access all Some_Record_T;
   type Some_Record_T is record
      Field : String (1 .. 10);
      Next  : Some_Record_Access_T;
   end record;

   Head : Some_Record_Access_T := null;
   Item : Some_Record_Access_T := null;

   Line : String (1 .. 10);
   Last : Natural;

 procedure Free is new Ada.Unchecked_Deallocation
   (Some_Record_T, Some_Record_Access_T);

begin

   loop
      Put ("Enter String: ");
      Get_Line (Line, Last);
      exit when Last = 0;
      Line (Last + 1 .. Line'last) := (others => ' ');
      Item                         := new Some_Record_T;
      Item.all                     := (Line, Head);
      Head                         := Item;
   end loop;

   Put_Line ("List");
   while Head /= null loop
      Put_Line ("  " & Head.Field);
      Head := Head.Next;
   end loop;

   Put_Line ("Delete");
   Free (Item);
   GNAT.Debug_Pools.Print_Info_Stdout (Storage_Pool);

end Simple;
```

Memory Debugging

GNAT.Debug_Pools

- Ada allows the coder to specify *where* the allocated memory comes from
  - Called *Storage Pool*
  - Basically, connecting **new** and `Unchecked_Deallocation` with some other code
  - More details in the next section

```
type Linked_List_Ptr_T is access all Linked_List_T;
for Linked_List_Ptr_T'storage_pool use Memory_Mgmt.Storage_Pool;
```

- GNAT uses this mechanism in the run-time package `GNAT.Debug_Pools` to track allocation/deallocation

```
with GNAT.Debug_Pools;
package Memory_Mgmt is
  Storage_Pool : GNAT.Debug_Pools.Debug_Pool;
end Memory_Mgmt;
```

GNAT.Debug_Pools Spec (Partial)

```
package GNAT.Debug_Pools is

   type Debug_Pool is new System.Checked_Pools.Checked_Pool with private;

   generic
      with procedure Put_Line (S : String) is <>;
      with procedure Put      (S : String) is <>;
   procedure Print_Info
     (Pool          : Debug_Pool;
      Cumulate      : Boolean := False;
      Display_Slots : Boolean := False;
      Display_Leaks : Boolean := False);

   procedure Print_Info_Stdout
     (Pool          : Debug_Pool;
      Cumulate      : Boolean := False;
      Display_Slots : Boolean := False;
      Display_Leaks : Boolean := False);
   --  Standard instantiation of Print_Info to print on standard_output.

   procedure Dump_Gnatmem (Pool : Debug_Pool; File_Name : String);
   --  Create an external file on the disk, which can be processed by gnatmem
   --  to display the location of memory leaks.

   procedure Print_Pool (A : System.Address);
   --  Given an address in memory, it will print on standard output the known
   --  information about this address

   function High_Water_Mark
     (Pool : Debug_Pool) return Byte_Count;
   --  Return the highest size of the memory allocated by the pool.

   function Current_Water_Mark
     (Pool : Debug_Pool) return Byte_Count;
   --  Return the size of the memory currently allocated by the pool.

private
   -- ...
end GNAT.Debug_Pools;
```

Displaying Debug Information

- Simple modifications to our linked list example
  - Create and use storage pool

```
with GNAT.Debug_Pools; -- Added
procedure Simple is
   Storage_Pool : GNAT.Debug_Pools.Debug_Pool; -- Added
   type Some_Record_T;
   type Some_Record_Access_T is access all Some_Record_T;
   for Some_Record_Access_T'storage_pool
       use Storage_Pool; -- Added
```

  - Dump info after each **new**

```
Item                            := new Some_Record_T;
GNAT.Debug_Pools.Print_Info_Stdout (Storage_Pool); -- Added
Item.all := (Line, Head);
```

  - Dump info after `free`

```
Free (Item);
GNAT.Debug_Pools.Print_Info_Stdout (Storage_Pool); -- Added
```

Execution Results

```
Enter String: X
Total allocated bytes :  24
Total logically deallocated bytes :  0
Total physically deallocated bytes :  0
Current Water Mark:  24
High Water Mark:  24

Enter String: Y
Total allocated bytes :  48
Total logically deallocated bytes :  0
Total physically deallocated bytes :  0
Current Water Mark:  48
High Water Mark:  48

Enter String:
List
  Y
  X
Delete
Total allocated bytes :  48
Total logically deallocated bytes :  24
Total physically deallocated bytes :  0
Current Water Mark:  24
High Water Mark:  48
```

Memory Control

System.Storage_Pools

- Mechanism to allow coder control over allocation/deallocation process
  - Uses `Ada.Finalization.Limited_Controlled` to implement customized memory allocation and deallocation.
  - Must be specified for each access type being controlled

```
type Boring_Access_T is access Some_T;
-- Storage Pools mechanism not used here
type Important_Access_T is access Some_T;
for Important_Access_T'storage_pool use My_Storage_Pool;
-- Storage Pools mechanism used for Important_Access_T
```

System.Storage_Pools Spec (Partial)

```
with Ada.Finalization;
with System.Storage_Elements;
package System.Storage_Pools with Pure is
  type Root_Storage_Pool is abstract
    new Ada.Finalization.Limited_Controlled with private;
  pragma Preelaborable_Initialization (Root_Storage_Pool);

  procedure Allocate
    (Pool                     : in out Root_Storage_Pool;
     Storage_Address          : out System.Address;
     Size_In_Storage_Elements : System.Storage_Elements.Storage_Count;
     Alignment                : System.Storage_Elements.Storage_Count)
  is abstract;

  procedure Deallocate
    (Pool                     : in out Root_Storage_Pool;
     Storage_Address          : System.Address;
     Size_In_Storage_Elements : System.Storage_Elements.Storage_Count;
     Alignment                : System.Storage_Elements.Storage_Count)
  is abstract;

  function Storage_Size
    (Pool : Root_Storage_Pool)
     return System.Storage_Elements.Storage_Count
  is abstract;

private
  -- ...
end System.Storage_Pools;
```

System.Storage_Pools Explanations

- Note `Root_Storage_Pool`, `Allocate`, `Deallocate`, and `Storage_Size` are **abstract**
  - You must create your own type derived from `Root_Storage_Pool`
  - You must create versions of `Allocate`, `Deallocate`, and `Storage_Size` to allocate/deallocate memory
- Parameters
  - `Pool`
    - Memory pool being manipulated
  - `Storage_Address`
    - For `Allocate` - location in memory where access type will point to
    - For `Deallocate` - location in memory where memory should be released
  - `Size_In_Storage_Elements`
    - Number of bytes needed to contain contents
  - `Alignment`
    - Byte alignment for memory location

System.Storage_Pools Example (Partial)

```
subtype Index_T is Storage_Count range 1 .. 1_000;
Memory_Block : aliased array (Index_T) of Interfaces.Unsigned_8;
Memory_Used  : array (Index_T) of Boolean := (others => False);

procedure Set_In_Use (Start  : Index_T;
                      Length : Storage_Count;
                      Used   : Boolean);

function Find_Free_Block (Length : Storage_Count) return Index_T;

procedure Allocate
  (Pool                     : in out Storage_Pool_T;
   Storage_Address          :    out System.Address;
   Size_In_Storage_Elements :        Storage_Count;
   Alignment                :        Storage_Count) is
   Index : Storage_Count := Find_Free_Block (Size_In_Storage_Elements);
begin
   Storage_Address := Memory_Block (Index)'address;
   Set_In_Use (Index, Size_In_Storage_Elements, True);
end Allocate;

procedure Deallocate
  (Pool                     : in out Storage_Pool_T;
   Storage_Address          :        System.Address;
   Size_In_Storage_Elements :        Storage_Count;
   Alignment                :        Storage_Count) is
begin
   for I in Memory_Block'range loop
      if Memory_Block (I)'address = Storage_Address then
         Set_In_Use (I, Size_In_Storage_Elements, False);
      end if;
   end loop;
end Deallocate;
```

Lab

Advanced Access Types Lab

- Build an application that adds / removes items from a linked list
    - At any time, user should be able to
        - Add a new item into the "appropriate" location in the list
        - Remove an item without changing the position of any other item in the list
        - Print the list
- This is a multi-step lab! First priority should be understanding linked lists, then, if you have time, storage pools
- Required goals
    1. Implement **Add** functionality
        - For this step, "appropriate" means either end of the list (but consistent - always front or always back)
    2. Implement **Print** functionality
    3. Implement **Delete** functionality

Extra Credit

- Complete as many of these as you have time for

  1. Use `GNAT.Debug_Pools` to print out the status of your memory allocation/deallocation after every **new** and `deallocate`
  2. Modify **Add** so that "appropriate" means in a sorted order
  3. Implement storage pools where you write your own memory allocation/deallocation routines

  - Should still be able to print memory status

Lab Solution - Database

```
with Ada.Strings.Unbounded;
package Database is
   type Database_T is private;
   function "=" (L, R : Database_T) return Boolean;
   function To_Database (Value : String) return Database_T;
   function From_Database (Value : Database_T) return String;
   function "<" (L, R : Database_T) return Boolean;
private
   type Database_T is record
      Value  : String (1 .. 100);
      Length : Natural;
   end record;
end Database;

package body Database is
   use Ada.Strings.Unbounded;
   function "=" (L, R : Database_T) return Boolean is
   begin
      return L.Value (1 .. L.Length) = R.Value (1 .. R.Length);
   end "=";
   function To_Database (Value : String) return Database_T is
      Retval : Database_T;
   begin
      Retval.Length                     := Value'length;
      Retval.Value (1 .. Retval.Length) := Value;
      return Retval;
   end To_Database;
   function From_Database (Value : Database_T) return String is
   begin
      return Value.Value (1 .. Value.Length);
   end From_Database;

   function "<" (L, R : Database_T) return Boolean is
   begin
      return L.Value (1 .. L.Length) < R.Value (1 .. R.Length);
   end "<";
end Database;
```

Lab Solution - Database_List (Spec)

```
with Database; use Database;
-- Uncomment next line when using debug/storage pools
-- with Memory_Mgmt;
package Database_List is
   type List_T is limited private;
   procedure First (List : in out List_T);
   procedure Next (List : in out List_T);
   function End_Of_List (List : List_T) return Boolean;
   function Current (List : List_T) return Database_T;
   procedure Insert (List    : in out List_T;
                 Element :        Database_T);
   procedure Delete (List    : in out List_T;
                  Element :        Database_T);
   function Is_Empty (List : List_T) return Boolean;
private
   type Linked_List_T;
   type Linked_List_Ptr_T is access all Linked_List_T;
   -- Uncomment next line when using debug/storage pools
   -- for Linked_List_Ptr_T'storage_pool use Memory_Mgmt.Storage_Pool;
   type Linked_List_T is record
      Next    : Linked_List_Ptr_T;
      Content : Database_T;
   end record;
   type List_T is record
      Head    : Linked_List_Ptr_T;
      Current : Linked_List_Ptr_T;
   end record;
end Database_List;
```

Lab Solution - Database_List (Helper Objects)

```
with Interfaces;
with Unchecked_Deallocation;
package body Database_List is
   use type Database.Database_T;

   function Is_Empty (List : List_T) return Boolean is
   begin
      return List.Head = null;
   end Is_Empty;

   procedure First (List : in out List_T) is
   begin
      List.Current := List.Head;
   end First;

   procedure Next (List : in out List_T) is
   begin
      if not Is_Empty (List) then
         if List.Current /= null then
            List.Current := List.Current.Next;
         end if;
      end if;
   end Next;

   function End_Of_List (List : List_T) return Boolean is
   begin
      return List.Current = null;
   end End_Of_List;

   function Current (List : List_T) return Database_T is
   begin
      return List.Current.Content;
   end Current;
```

Lab Solution - Database_List (Insert/Delete)

```
   procedure Insert (List    : in out List_T;
                     Element :        Database_T) is
      New_Element : Linked_List_Ptr_T :=
        new Linked_List_T'(Next => null, Content => Element);
   begin
      if Is_Empty (List) then
         List.Current := New_Element;
         List.Head    := New_Element;
      elsif Element < List.Head.Content then
         New_Element.Next := List.Head;
         List.Current     := New_Element;
         List.Head        := New_Element;
      else
         declare
            Current : Linked_List_Ptr_T := List.Head;
         begin
            while Current.Next /= null and then Current.Next.Content < Element
            loop
               Current := Current.Next;
            end loop;
            New_Element.Next := Current.Next;
            Current.Next     := New_Element;
         end;
      end if;
      -- Uncomment next line when using debug/storage pools
      -- Memory_Mgmt.Print_Info;
   end Insert;

   procedure Free is new Unchecked_Deallocation
     (Linked_List_T, Linked_List_Ptr_T);
   procedure Delete
     (List    : in out List_T;
      Element :        Database_T) is
      To_Delete : Linked_List_Ptr_T := null;
   begin
      if not Is_Empty (List) then
         if List.Head.Content = Element then
            To_Delete    := List.Head;
            List.Head    := List.Head.Next;
            List.Current := List.Head;
         else
            declare
               Previous : Linked_List_Ptr_T := List.Head;
               Current  : Linked_List_Ptr_T := List.Head.Next;
            begin
               while Current /= null loop
                  if Current.Content = Element then
                     To_Delete     := Current;
                     Previous.Next := Current.Next;
                  end if;
                  Current := Current.Next;
               end loop;
            end;
            List.Current := List.Head;
         end if;
         if To_Delete /= null then
            Free (To_Delete);
         end if;
      end if;
      -- Uncomment next line when using debug/storage pools
      -- Memory_Mgmt.Print_Info;
   end Delete;
end Database_List;
```

Lab Solution - Main

```
with Simple_Io; use Simple_Io;
with Database;
with Database_List;
procedure Main is
   List    : Database_List.List_T;
   Element : Database.Database_T;

   procedure Add is
      Value : constant String := Get_String ("Add");
   begin
      if Value'length > 0 then
         Element := Database.To_Database (Value);
         Database_List.Insert (List, Element);
      end if;
   end Add;

   procedure Delete is
      Value : constant String := Get_String ("Delete");
   begin
      if Value'length > 0 then
         Element := Database.To_Database (Value);
         Database_List.Delete (List, Element);
      end if;
   end Delete;

   procedure Print is
   begin
      Database_List.First (List);
      Simple_Io.Print_String ("List");
      while not Database_List.End_Of_List (List) loop
         Element := Database_List.Current (List);
         Print_String ("  " & Database.From_Database (Element));
         Database_List.Next (List);
      end loop;
   end Print;

begin
   loop
      case Get_Character ("A=Add D=Delete P=Print Q=Quit") is
         when 'a' | 'A' => Add;
         when 'd' | 'D' => Delete;
         when 'p' | 'P' => Print;
         when 'q' | 'Q' => exit;
         when others    => null;
      end case;
   end loop;
end Main;
```

Lab Solution - Simple_IO (Spec)

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Simple_Io is
   function Get_String (Prompt : String)
                        return String;
   function Get_Number (Prompt : String)
                        return Integer;
   function Get_Character (Prompt : String)
                           return Character;
   procedure Print_String (Str : String);
   procedure Print_Number (Num : Integer);
   procedure Print_Character (Char : Character);
   function Get_String (Prompt : String)
                        return Unbounded_String;
   procedure Print_String (Str : Unbounded_String);
end Simple_Io;
```

Lab Solution - Simple_IO (Body)

```
with Ada.Text_IO;
package body Simple_Io is
   function Get_String (Prompt : String) return String is
      Str  : String (1 .. 1_000);
      Last : Integer;
   begin
      Ada.Text_IO.Put (Prompt & "> ");
      Ada.Text_IO.Get_Line (Str, Last);
      return Str (1 .. Last);
   end Get_String;

   function Get_Number (Prompt : String) return Integer is
      Str : constant String := Get_String (Prompt);
   begin
      return Integer'value (Str);
   end Get_Number;

   function Get_Character (Prompt : String) return Character is
      Str : constant String := Get_String (Prompt);
   begin
      return Str (Str'first);
   end Get_Character;

   procedure Print_String (Str : String) is
   begin
      Ada.Text_IO.Put_Line (Str);
   end Print_String;
   procedure Print_Number (Num : Integer) is
   begin
      Ada.Text_IO.Put_Line (Integer'image (Num));
   end Print_Number;
   procedure Print_Character (Char : Character) is
   begin
      Ada.Text_IO.Put_Line (Character'image (Char));
   end Print_Character;

   function Get_String (Prompt : String) return Unbounded_String is
   begin
      return To_Unbounded_String (Get_String (Prompt));
   end Get_String;
   procedure Print_String (Str : Unbounded_String) is
   begin
      Print_String (To_String (Str));
   end Print_String;
end Simple_Io;
```

Lab Solution - Memory_Mgmt (Debug Pools)

```
with GNAT.Debug_Pools;
package Memory_Mgmt is
   Storage_Pool : GNAT.Debug_Pools.Debug_Pool;
   procedure Print_Info;
end Memory_Mgmt;

package body Memory_Mgmt is
   procedure Print_Info is
   begin
      GNAT.Debug_Pools.Print_Info_Stdout (Storage_Pool);
   end Print_Info;
end Memory_Mgmt;
```

Lab Solution - Memory_Mgmt (Storage Pools Spec)

```
with System.Storage_Elements;
with System.Storage_Pools;
package Memory_Mgmt is

   type Storage_Pool_T is new System.Storage_Pools.Root_Storage_Pool with
   null record;

   procedure Print_Info;

   procedure Allocate
     (Pool                     : in out Storage_Pool_T;
      Storage_Address          :    out System.Address;
      Size_In_Storage_Elements :        System.Storage_Elements.Storage_Count;
      Alignment                :        System.Storage_Elements.Storage_Count);
   procedure Deallocate
     (Pool                     : in out Storage_Pool_T;
      Storage_Address          :        System.Address;
      Size_In_Storage_Elements :        System.Storage_Elements.Storage_Count;
      Alignment                :        System.Storage_Elements.Storage_Count);
   function Storage_Size
     (Pool : Storage_Pool_T)
      return System.Storage_Elements.Storage_Count;

   Storage_Pool : Storage_Pool_T;

end Memory_Mgmt;
```

Lab Solution - Memory_Mgmt (Storage Pools 1/2)

```
with Ada.Text_IO;
with Interfaces;
package body Memory_Mgmt is
   use System.Storage_Elements;
   use type System.Address;

   subtype Index_T is Storage_Count range 1 .. 1_000;
   Memory_Block : aliased array (Index_T) of Interfaces.Unsigned_8;
   Memory_Used  : array (Index_T) of Boolean := (others => False);

   Current_Water_Mark : Storage_Count := 0;
   High_Water_Mark    : Storage_Count := 0;

   procedure Set_In_Use
     (Start  : Index_T;
      Length : Storage_Count;
      Used   : Boolean) is
   begin
      for I in 0 .. Length - 1 loop
         Memory_Used (Start + I) := Used;
      end loop;
      if Used then
         Current_Water_Mark := Current_Water_Mark + Length;
         High_Water_Mark    :=
           Storage_Count'max (High_Water_Mark, Current_Water_Mark);
      else
         Current_Water_Mark := Current_Water_Mark - Length;
      end if;
   end Set_In_Use;

   function Find_Free_Block
     (Length : Storage_Count)
      return Index_T is
      Consecutive : Storage_Count := 0;
   begin
      for I in Memory_Used'range loop
         if Memory_Used (I) then
            Consecutive := 0;
         else
            Consecutive := Consecutive + 1;
            if Consecutive >= Length then
               return I;
            end if;
         end if;
      end loop;
      raise Storage_Error;
   end Find_Free_Block;
```

Lab Solution - Memory_Mgmt (Storage Pools 2/2)

```
   procedure Allocate
     (Pool                     : in out Storage_Pool_T;
      Storage_Address          :    out System.Address;
      Size_In_Storage_Elements :        Storage_Count;
      Alignment                :        Storage_Count) is
      Index : Storage_Count := Find_Free_Block (Size_In_Storage_Elements);
   begin
      Storage_Address := Memory_Block (Index)'address;
      Set_In_Use (Index, Size_In_Storage_Elements, True);
   end Allocate;

   procedure Deallocate
     (Pool                     : in out Storage_Pool_T;
      Storage_Address          :        System.Address;
      Size_In_Storage_Elements :        Storage_Count;
      Alignment                :        Storage_Count) is
   begin
      for I in Memory_Block'range loop
         if Memory_Block (I)'address = Storage_Address then
            Set_In_Use (I, Size_In_Storage_Elements, False);
         end if;
      end loop;
   end Deallocate;

   function Storage_Size
     (Pool : Storage_Pool_T)
      return System.Storage_Elements.Storage_Count is
   begin
      return 0;
   end Storage_Size;

   procedure Print_Info is
   begin
      Ada.Text_IO.Put_Line
        ("Current Water Mark: " & Storage_Count'image (Current_Water_Mark));
      Ada.Text_IO.Put_Line
        ("High Water Mark: " & Storage_Count'image (High_Water_Mark));
   end Print_Info;

end Memory_Mgmt;
```

Summary

Summary

- Access types when used with "dynamic" memory allocation can cause problems
  - Whether actually dynamic or using managed storage pools, memory leaks/lack can occur
  - Storage pools can help diagnose memory issues, but it's still a usage issue
- `GNAT.Debug_Pools` is useful for debugging memory issues
  - Mostly in low-level testing
  - Could integrate it with an error logging mechanism
- `System.Storage_Pools` can be used to control memory usage
  - Adds overhead

Genericity

Introduction

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;
begin
   V := Left;
   Left := Right;
   Right := V;
end Swap_Int;

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

Solution: Generics

- A generic unit is a unit that does not exist
- It is a pattern based on properties
- The instantiation applies the pattern to certain parameters

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

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

Creating Generics

What Can Be Made Generic?

- Subprograms and packages can be made generic

```
generic
   type T is private;
procedure Swap (L, R : in out T)
generic
   type T is private;
package Stack is
   procedure Push ( Item : T );
   ...
```

- Children of generic units have to be generic themselves

```
generic
package Stack.Utilities is
   procedure Print is
```

How Do You Use A Generic?

- Generic instantiation is creating new set of data where a generic package contains library-level variables:

```
package Integer_stack is new Stack ( Integer );
package Integer_Stack_Utils is
   new Integer_Stack.Utilities;
...
Integer_Stack.Push ( 1 );
Integer_Stack_Utils.Print;
```

Generic Data

Examples

```
package Generic_Data is
   generic
      type Discrete_T is (<>);
      type Integer_T is range <>;
      type Float_T is digits <>;
      type Indefinite_T;
      type Tagged_T is tagged;
      type Array_T is array (Boolean) of Integer;
      type Access_T is access all Integer;
      type Private_T is private;
      type Unconstrained_T (<>) is private;
   package Parameter_Properties is
      procedure Do_Something ( Discrete_Param      : Discrete_T;
                               Integer_Param       : Integer_T;
                               Float_Param         : Float_T;
                               Indefinite_Param    : access Indefinite_T;
                               Tagged_Param        : Tagged_T;
                               Array_Param         : Array_T;
                               Access_Param        : Access_T;
                               Private_Param       : Private_T;
                               Unconstrained_Param : Unconstrained_T);
   end Parameter_Properties;

   generic
      type Item_T is private;
      type Access_Item_T is access all Item_T;
      type Index_T is (<>);
      type Array_T is array (Index_T range <>) of Access_Item_T;
   package Combination is
      procedure Add (List  : in out Array_T;
                     Index : in     Index_T;
                     Item  : in     Item_T);
   end Combination;
end Generic_Data;

with Types; use Types;
with Generic_Data;
package Generic_Instances is
   package Parameter_Properties_Instance is new Generic_Data
     .Parameter_Properties
     (Boolean, Integer, Float, Indefinite_T => Hidden_T,
      Tagged_T => Tagged_Record_T, Array_T => Boolean_Array_Of_Integers_T,
      Access_T => Access_Integer_T, Private_T => Some_Private_T,
      Unconstrained_T                        => String);

   type Item_T is (Red, White, Blue);
   type Access_T is access all Item_T;
   type Index_T is range 1 .. 100;
   type Array_T is array (Index_T range <>) of Access_T;
   package Combination_Instance is new Generic_Data.Combination
     (Item_T, Access_T, Index_T, Array_T);
end Generic_Instances;

package body Generic_Data is
   package body Parameter_Properties is
      procedure Do_Something ( Discrete_Param      : Discrete_T;
                               Integer_Param       : Integer_T;
                               Float_Param         : Float_T;
                               Indefinite_Param    : access Indefinite_T;
                               Tagged_Param        : Tagged_T;
                               Array_Param         : Array_T;
                               Access_Param        : Access_T;
                               Private_Param       : Private_T;
                               Unconstrained_Param : Unconstrained_T) is null;
   end Parameter_Properties;

   package body Combination is
      procedure Add (List  : in out Array_T;
                     Index : in     Index_T;
                     Item  : in     Item_T) is
      begin
         List (Index) := new Item_T'(Item);
      end Add;
   end Combination;
end Generic_Data;

package Types is
   type Hidden_T;
   type Tagged_Record_T is tagged record
      Field : access Hidden_T;
   end record;
   type Hidden_T is private;
   type Boolean_Array_Of_Integers_T is array (Boolean) of Integer;
   type Access_Integer_T is access all Integer;
   type Some_Private_T is private;
private
   type Hidden_T is new Integer;
   type Some_Private_T is new Integer;
end Types;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/160_genericity.html#generic-data

Generic Types Parameters (1/2)

- A generic parameter is a template
- It specifies the properties the generic body can rely on

```
generic
   type T1 is private; -- should have properties
                       -- of private type (assignment,
                       -- comparison, able to declare
                       -- variables on the stack...)
   type T2 (<>) is private;    -- can be unconstrained
   type T3 is limited private; -- can be limited
package Parent is [...]
```

- The actual parameter must provide at least as many properties as the generic contract

Generic Types Parameters (2/2)

- The usage in the generic has to follow the contract

```
generic
   type T (<>) is private;
procedure P (V : T);
procedure P (V : T) is
   X1 : T := V; -- OK, can constrain by initialization
   X2 : T;      -- Compilation error, no constraint to this
begin
...
type L_T is limited null record;
...
-- unconstrained types are accepted
procedure P1 is new P (String);
-- type is already constrained
procedure P2 is new P (Integer);
-- Illegal: the type can't be limited because the generic
-- is allowed to make copies
procedure P3 is new P (L_T);
```

Possible Properties for Generic Types

```
type T1 is (<>); -- discrete
type T2 is range <>; -- integer
type T3 is digits <>; -- float
type T4 (<>); -- indefinite
type T5 is tagged;
type T6 is array ( Boolean ) of Integer;
type T7 is access integer;
type T8 (<>) is [limited] private;
```

Generic Parameters Can Be Combined

- Consistency is checked at compile-time

```
generic
   type T (<>) is limited private;
   type Acc is access all T;
   type Index is (<>);
   type Arr is array (Index range <>) of Acc;
procedure P;

type String_Ptr is access all String;
type String_Array is array (Integer range <>)
    of String_Ptr;

procedure P_String is new P
   (T     => String,
    Acc   => String_Ptr,
    Index => Integer,
    Arr   => String_Array);
```

Quiz

```
generic
   type T is tagged;
   type T2;
procedure G_P;

type Tag is tagged null record;
type Arr is array (Positive range <>) of Tag;
```

Which declaration(s) is(are) legal?

A. `procedure P is new G_P (Tag, Arr)`
B. `procedure P is new G_P (Arr, Tag)`
C. `procedure P is new G_P (Tag, Tag)`
D. `procedure P is new G_P (Arr, Arr)`

Quiz

```
generic
   type T is tagged;
   type T2;
procedure G_P;

type Tag is tagged null record;
type Arr is array (Positive range <>) of Tag;
```

Which declaration(s) is(are) legal?

A. *procedure P is new G_P (Tag, Arr)*
B. procedure P is new G_P (Arr, Tag)
C. *procedure P is new G_P (Tag, Tag)*
D. procedure P is new G_P (Arr, Arr)

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is an illegal instantiation?

A. `procedure A is new G (String, Character);`
B. `procedure B is new G (Character, Integer);`
C. `procedure C is new G (Integer, Boolean);`
D. `procedure D is new G (Boolean, String);`

Quiz

```
generic
   type T1 is (<>);
   type T2 (<>) is private;
procedure G
  (A : T1;
   B : T2);
```

Which is an illegal instantiation?

A. *`procedure A is new G (String, Character);`*
B. `procedure B is new G (Character, Integer);`
C. `procedure C is new G (Integer, Boolean);`
D. `procedure D is new G (Boolean, String);`

T1 must be discrete - so an integer or an enumeration. T2 can be any type

Generic Formal Data

Examples

```ada
package Generic_Formal_Data is
   generic
      type Variable_T is range <>;
      Variable : in out Variable_T;
      Increment : Variable_T;
   package Constants_And_Variables is
      procedure Add;
      function Value return Variable_T is (Variable);
   end Constants_And_Variables;

   generic
      type Type_T is (<>);
      with procedure Print_One (Prompt : String; Value : Type_T);
      with procedure Print_Two (Prompt : String; Value : Type_T) is null;
      with procedure Print_Three (Prompt : String; Value : Type_T) is <>;
   package Subprogram_Parameters is
      procedure Print (Prompt : String; Param : Type_T);
   end Subprogram_Parameters;
end Generic_Formal_Data;

with Ada.Text_IO;         use Ada.Text_IO;
with Generic_Formal_Data; use Generic_Formal_Data;
procedure Test_Generic_Formal_Data is
   procedure Print_One (Prompt : String; Param : Integer) is
   begin
      Put_Line (Prompt & " - Print_One" & Param'Image);
   end Print_One;
   procedure Print_Two (Prompt : String; Param : Integer) is
   begin
      Put_Line (Prompt & " - Print_Two" & Param'Image);
   end Print_Two;
   procedure Print_Three (Prompt : String; Param : Integer) is
   begin
      Put_Line (Prompt & " - Print_Three" & Param'Image);
   end Print_Three;
   procedure Print_Three_Prime (Prompt : String; Param : Integer) is
   begin
      Put_Line (Prompt & " - Print_Three_Prime" & Param'Image);
   end Print_Three_Prime;

   Global_Object : Integer := 0;
   package Global_Data is new Constants_And_Variables
     (Integer, Global_Object, 111);

   package Print_1 is new Subprogram_Parameters (Integer, Print_One);
   package Print_2 is new Subprogram_Parameters (Integer, Print_One, Print_Two);
   package Print_3 is new Subprogram_Parameters (Integer, Print_One, Print_Two, Print_Three_Prime);

begin
   Print_1.Print ("print_1", Global_Data.Value);
   Global_Data.Add;
   Print_2.Print ("print_2", Global_Data.Value);
   Global_Data.Add;
   Print_3.Print ("print_3", Global_Data.Value);
end Test_Generic_Formal_Data;

package body Generic_Formal_Data is
   package body Constants_And_Variables is
      procedure Add is
      begin
         Variable := Variable + Increment;
      end Add;
   end Constants_And_Variables;

   package body Subprogram_Parameters is
      procedure Print (Prompt : String; Param : Type_T) is
      begin
         Print_One (Prompt, Param);
         Print_Two (Prompt, Param);
         Print_Three (Prompt, Param);
      end Print;
   end Subprogram_Parameters;
end Generic_Formal_Data;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/160_genericity.html#generic-formal-data

Generic 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);
```

Generic Subprogram Parameters Defaults

Ada 2005

- `is <>` - matching subprogram is taken by default
- `is null` - null subprogram is taken by default
  - Only available in Ada 2005 and later

```
generic
  with procedure Callback1 is <>;
  with procedure Callback2 is null;
procedure P;
procedure Callback1;
procedure P_I is new P;
-- takes Callback1 and null
```

Generic Package Parameters

- A generic unit can depend on the instance of another generic unit
- Parameters of the instantiation can be constrained partially or completely

```
generic
   type T1 is private;
   type T2 is private;
package Base is [...]

generic
   with package B is new Base (Integer, <>);
   V : B.T2;
package Other [...]

package Base_I is new Base (Integer, Float);

package Other_I is new Other (Base_I, 56.7);
```

Quiz

```
generic
   type T is (<>);
   G_A : in out T;
procedure G_P;

type I is new Integer;
type E is (OK, NOK);
type F is new Float;
X : I;
Y : E;
Z : F;
```

A. `procedure P is new G_P (I, X)`
B. `procedure P is new G_P (E, Y)`
C. `procedure P is new G_P (I, E'Pos (Y))`
D. `procedure P is new G_P (F, Z)`

Quiz

```
generic
   type T is (<>);
   G_A : in out T;
procedure G_P;

type I is new Integer;
type E is (OK, NOK);
type F is new Float;
X : I;
Y : E;
Z : F;
```

A. *procedure P is new G_P (I, X)*
B. *procedure P is new G_P (E, Y)*
C. procedure P is new G_P (I, E'Pos (Y))
D. procedure P is new G_P (F, Z)

Quiz

```
generic
   type L is limited private;
   type P is private;
procedure G_P;

type Lim is limited null record;
type Int is new Integer;

type Rec is record
   L : Lim;
   I : Int;
end record;
```

Which declaration(s) is(are) legal?

A. `procedure P is new G_P (Lim, Int)`
B. `procedure P is new G_P (Int, Rec)`
C. `procedure P is new G_P (Rec, Rec)`
D. `procedure P is new G_P (Int, Int)`

Quiz

```
generic
   type L is limited private;
   type P is private;
procedure G_P;

type Lim is limited null record;
type Int is new Integer;

type Rec is record
   L : Lim;
   I : Int;
end record;
```

Which declaration(s) is(are) legal?

A. *procedure P is new G_P (Lim, Int)*
B. procedure P is new G_P (Int, Rec)
C. procedure P is new G_P (Rec, Rec)
D. *procedure P is new G_P (Int, Int)*

Quiz

Ada 2005

```
1  procedure P1 (X : in out Integer); -- add 100 to X
2  procedure P2 (X : in out Integer); -- add 20 to X
3  procedure P3 (X : in out Integer); -- add 3 to X
4  generic
5     with procedure P1 (X : in out Integer) is <>;
6     with procedure P2 (X : in out Integer) is null;
7  procedure G ( P : integer );
8  procedure G ( P : integer ) is
9     X : integer := P;
10 begin
11    P1(X);
12    P2(X);
13    Ada.Text_IO.Put_Line ( X'Image );
14 end G;
15 procedure Instance is new G ( P1 => P3 );
```

What is printed when `Instance` is called?

A. 100
B. 120
C. 3
D. 103

Explanations

A. Wrong - result for
`procedure Instance is new G;`
B. Wrong - result for
`procedure Instance is new G(P1,P2);`
C. P1 at line 12 is mapped to P3 at line 3, and P2 at line 14 wasn't specified so it defaults to `null`
D. Wrong - result for
`procedure Instance is new G(P2=>P3);`

Quiz

Ada 2005

```
1   procedure P1 (X : in out Integer); -- add 100 to X
2   procedure P2 (X : in out Integer); -- add 20 to X
3   procedure P3 (X : in out Integer); -- add 3 to X
4   generic
5      with procedure P1 (X : in out Integer) is <>;
6      with procedure P2 (X : in out Integer) is null;
7   procedure G ( P : integer );
8   procedure G ( P : integer ) is
9      X : integer := P;
10  begin
11     P1(X);
12     P2(X);
13     Ada.Text_IO.Put_Line ( X'Image );
14  end G;
15  procedure Instance is new G ( P1 => P3 );
```

What is printed when `Instance` is called?

A. 100
B. 120
C. ***3***
D. 103

Explanations

A. Wrong - result for `procedure Instance is new G;`
B. Wrong - result for `procedure Instance is new G(P1,P2);`
C. P1 at line 12 is mapped to P3 at line 3, and P2 at line 14 wasn't specified so it defaults to `null`
D. Wrong - result for `procedure Instance is new G(P2=>P3);`

Generic Completion

Implications at Compile-Time

- The body needs to be visible when compiling the user code
- Therefore, when distributing a component with generics to be instantiated, the code of the generic must come along

Generic and Freezing Points

- A generic type **freezes** the type and needs the **full view**
- May force separation between its declaration (in spec) and instantiations (in private or body)

```
generic
   type X is private;
package Base is
   V : access X;
end Base;

package P is
   type X is private;
   -- illegal
   package B is new Base (X);
private
   type X is null record;
end P;
```

Generic Incomplete Parameters

- A generic type can be incomplete
- Allows generic instantiations before full type definition
- Restricts the possible usages (only `access`)

```
generic
   type X; -- incomplete
package Base is
   V : access X;
end Base;

package P is
   type X is private;
   -- legal
   package B is new Base (X);
private
   type X is null record;
end P;
```

Quiz

```
generic
   type T1;
   A1 : access T1;
   type T2 is private;
   A2, B2 : T2;
procedure G_P;
procedure G_P is
begin
   -- Complete here
end G_P;
```

Which of the following statement(s) is(are) valid for P's body?

A. `pragma Assert (A1 /= null)`
B. `pragma Assert (A1.all'Size > 32)`
C. `pragma Assert (A2 = B2)`
D. `pragma Assert (A2 - B2 /= 0)`

Quiz

```
generic
   type T1;
   A1 : access T1;
   type T2 is private;
   A2, B2 : T2;
procedure G_P;
procedure G_P is
begin
   -- Complete here
end G_P;
```

Which of the following statement(s) is(are) valid for P's body?

A. *`pragma Assert (A1 /= null)`*
B. `pragma Assert (A1.all'Size > 32)`
C. *`pragma Assert (A2 = B2)`*
D. `pragma Assert (A2 - B2 /= 0)`

Lab

Genericity Lab

- Requirements
  - Create a list ADT to hold any type of data
    - Operations should include adding to the list and sorting the list
  - Create a record structure containing multiple fields
  - The **main** program should:
    - Allow the addition of multiple records into the list
    - Sort the list
    - Print the list
- Hints
  - Sort routine will need to know how to compare elements

Genericity Lab Solution - Generic (Spec)

```
generic
  type Element_T is private;
  Max_Size : Natural;
  with function "<" (L, R : Element_T) return Boolean is <>;
package Generic_List is

  type List_T is tagged private;

  procedure Add (This : in out List_T;
                 Item : in     Element_T);
  procedure Sort (This : in out List_T);

private
  subtype Index_T is Natural range 0 .. Max_Size;
  type List_Array_T is array (1 .. Index_T'Last) of Element_T;

  type List_T is tagged record
    Values : List_Array_T;
    Length : Index_T := 0;
  end record;
end Generic_List;
```

Genericity Lab Solution - Generic (Body)

```
package body Generic_List is

  procedure Add (This : in out List_T;
                 Item : in     Element_T) is
  begin
    This.Length                := This.Length + 1;
    This.Values (This.Length) := Item;
  end Add;

  procedure Sort (This : in out List_T) is
    Temp : Element_T;
  begin
    for I in 1 .. This.Length loop
      for J in I + 1 .. This.Length loop
        if This.Values (J) < This.Values (J - 1) then
          Temp                 := This.Values (J);
          This.Values (J)      := This.Values (J - 1);
          This.Values (J - 1) := Temp;
        end if;
      end loop;
    end loop;
  end Sort;
end Generic_List;
```

Genericity Lab Solution - Generic Output

```
generic
  with function Image (Element : Element_T) return String;
package Generic_List.Output is
  procedure Print (List : List_T);
end Generic_List.Output;

with Ada.Text_IO; use Ada.Text_IO;
package body Generic_List.Output is
  procedure Print (List : List_T) is
  begin
    for I in 1 .. List.Length loop
      Put_Line (Integer'Image (I) & ") " &
                Image (List.Values (I)));
    end loop;
  end Print;
end Generic_List.Output;
```

Genericity Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Data_Type;
with Generic_List;
with Generic_List.Output;
use type Data_Type.Record_T;
procedure Main is
  package List is new Generic_List (Data_Type.Record_T, 10);
  package Output is new List.Output (Data_Type.Image);

  My_List : List.List_T;
  Element : Data_Type.Record_T;

begin
  loop
    Put ("Enter character: ");
    declare
      Str : constant String := Get_Line;
    begin
      exit when Str'Length = 0;
      Element.Field2 := Str (1);
    end;
    Put ("Enter number: ");
    declare
      Str : constant String := Get_Line;
    begin
      exit when Str'Length = 0;
      Element.Field1 := Integer'Value (Str);
    end;
    My_List.Add (Element);
  end loop;

  My_List.Sort;
  Output.Print (My_List);
end Main;
```

Summary

Generic Routines vs Common Routines

```
package Helper is
   type Float_T is digits 6;
   generic
      type Type_T is digits <>;
      Min : Type_T;
      Max : Type_T;
   function In_Range_Generic (X : Type_T) return Boolean;
   function In_Range_Common (X   : Float_T;
                             Min : Float_T;
                             Max : Float_T)
                             return Boolean;
end Helper;

procedure User is
  type Speed_T is new Float_T range 0.0 .. 100.0;
  B : Boolean;
  function Valid_Speed is new In_Range_Generic
     (Speed_T, Speed_T'First, Speed_T'Last);
begin
  B := Valid_Speed (12.3);
  B := In_Range_Common (12.3, Speed_T'First, Speed_T'Last);
```

Summary

- Generics are useful for copying code that works the same just for different types
  - Sorting, containers, etc
- Properly written generics only need to be tested once
  - But testing / debugging can be more difficult
- Generic instantiations are best done at compile time
  - At the package level
  - Can be run-time expensive when done in subprogram scope

Tagged Derivation

Introduction

Object-Oriented Programming With Tagged Types

- For `record` types

```
type T is tagged record
...
```

- Child types can add new components (*attributes*)
- Object of a child type can be **substituted** for base type
- Primitive (*method*) can **dispatch at runtime** depending on the type at call-site
- Types can be **extended** by other packages
  - Casting and qualification to base type is allowed
- Private data is encapsulated through **privacy**

Tagged 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

Examples

```
package Tagged_Derivation is

   type Root_T is tagged record
      Root_Field : Integer;
   end record;
   function Primitive_1 (This : Root_T) return Integer is (This.Root_Field);
   function Primitive_2 (This : Root_T) return String is
     (Integer'Image (This.Root_Field));

   type Child_T is new Root_T with record
      Child_Field : Integer;
   end record;
   overriding function Primitive_2 (This : Child_T) return String is
     (Integer'Image (This.Root_Field) & " " &
      Integer'Image (This.Child_Field));
   function Primitive_3 (This : Child_T) return Integer is
     (This.Root_Field + This.Child_Field);

   -- type Simple_Deriviation_T is new Child_T; -- illegal

   type Root2_T is tagged record
      Root_Field : Integer;
   end record;
   -- procedure Primitive_4 (X : Root_T; Y : Root2_T); -- illegal

end Tagged_Derivation;

with Ada.Text_IO;       use Ada.Text_IO;
with Tagged_Derivation; use Tagged_Derivation;
procedure Test_Tagged_Derivation is
   Root  : Root_T  := (Root_Field => 1);
   Child : Child_T := (Root_Field => 11, Child_Field => 22);
begin
   Put_Line ("Root: " & Primitive_2 (Root));
   Put_Line ("Child: " & Primitive_2 (Child));
   Root := Root_T (Child);
   Put_Line ("Root from Child: " & Primitive_2 (Root));
   -- Child := Child_T ( Root ); -- illegal
   -- Put_Line ( "Child from Root: " & Primitive_2 ( Child ) ); -- illegal
   Child := (Root with Child_Field => 999);
   Put_Line ("Child from Root via aggregate: " & Primitive_2 (Child));
end Test_Tagged_Derivation;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/170_tagged_derivation.html#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
```

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

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

Primitives

- Child **cannot remove** a primitive
- Child **can add** new primitives
- **Controlling parameter**
  - Parameters the subprogram is a primitive of
  - For tagged types, all should have the **same type**

```
type Root1 is tagged null record;
type Root2 is tagged null record;

procedure P1 (V1 : Root1;
              V2 : Root1);
procedure P2 (V1 : Root1;
              V2 : Root2); -- illegal
```

Freeze Point For Tagged Types

- Freeze point definition does not change
  - A variable of the type is declared
  - The type is derived
  - The end of the scope is reached
- Declaring tagged type primitives past freeze point is **forbidden**

```
type Root is tagged null record;

procedure Prim (V : Root);

type Child is new Root with null record; -- freeze root

procedure Prim2 (V : Root); -- illegal

V : Child; --  freeze child

procedure Prim3 (V : Child); -- illegal
```

Tagged Aggregate

- At initialization, all fields (including **inherited**) must have a **value**

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

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

V : Child := (F1 => 0, F2 => 0);
```

- For **private types** use **aggregate extension**
  - Copy of a parent instance
  - Use `with null record` absent new fields

```
V2 : Child := (Parent_Instance with F2 => 0);
V3 : Empty_Child := (Parent_Instance with null record);
```

Overriding Indicators

Ada 2005

- Optional `overriding` and `not overriding` indicators

```
type Root is tagged null record;

procedure Prim1 (V : Root);
procedure Prim2 (V : Root);

type Child is new Root with null record;

overriding procedure Prim1 (V : Child);
-- Prim2 (V : Child) is implicitely inherited
not overriding procedure Prim3 (V : Child);
```

Prefix Notation

Ada 2012

- 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;
```

Quiz

Which declaration(s) will make `P` a primitive of `T1`?

A.
```
type T1 is tagged null record;
procedure P (O : T1) is null;
```

B.
```
type T0 is tagged null record;
type T1 is new T0 with null record;
type T2 is new T0 with null record;
procedure P (O : T1) is null;
```

C.
```
type T1 is tagged null record;
generic
  type T is tagged private;
package G_Pkg is
  type T2 is new T with null record;
end G_Pkg;
package Pkg is new G_Pkg (T1);
procedure P (O : T1) is null;
```

D.
```
type T1 is tagged null record;
generic
  type T;
procedure G_P (O : T);
procedure G_P (O : T) is null;
procedure P is new G_P (T1);
```

Quiz

Which declaration(s) will make P a primitive of T1?

A.
```
type T1 is tagged null record;
procedure P (O : T1) is null;
```

B.
```
type T0 is tagged null record;
type T1 is new T0 with null record;
type T2 is new T0 with null record;
procedure P (O : T1) is null;
```

C.
```
type T1 is tagged null record;
generic
  type T is tagged private;
package G_Pkg is
  type T2 is new T with null record;
end G_Pkg;
package Pkg is new G_Pkg (T1);
procedure P (O : T1) is null;
```

D.
```
type T1 is tagged null record;
generic
  type T;
procedure G_P (O : T);
procedure G_P (O : T) is null;
procedure P is new G_P (T1);
```

Quiz

```
with Pkg1; -- Defines tagged type Tag1, with primitive P
with Pkg2; use Pkg2; -- Defines tagged type Tag2, with primitive P
with Pkg3; -- Defines tagged type Tag3, with primitive P
use type Pkg3.Tag3;

procedure Main is
   O1 : Pkg1.Tag1;
   O2 : Pkg2.Tag2;
   O3 : Pkg3.Tag3;
```

Which statement(s) is(are) valid?

A. `O1.P`
B. `P (O1)`
C. `P (O2)`
D. `P (O3)`

Quiz

```
with Pkg1; -- Defines tagged type Tag1, with primitive P
with Pkg2; use Pkg2; -- Defines tagged type Tag2, with primitive P
with Pkg3; -- Defines tagged type Tag3, with primitive P
use type Pkg3.Tag3;

procedure Main is
   O1 : Pkg1.Tag1;
   O2 : Pkg2.Tag2;
   O3 : Pkg3.Tag3;
```

Which statement(s) is(are) valid?

A. *O1.P*
B. P (O1)
C. *P (O2)*
D. P (O3)

D. Only operators are use`d, should have been :ada:`use all

Quiz

Which code block is legal?

A.
```
type A1 is record
  Field1 : Integer;
end record;
type A2 is new A1 with
null record;
```

B.
```
type B1 is tagged
record
  Field2 :  Integer;
end record;
type B2 is new B1 with
record
  Field2b :  Integer;
end record;
```

C.
```
type C1 is tagged
record
  Field3 : Integer;
end record;
type C2 is new C1 with
record
  Field3 : Integer;
end record;
```

D.
```
type D1 is tagged
record
  Field1 : Integer;
end record;
type D2 is new D1;
```

Quiz

Which code block is legal?

A.
```
type A1 is record
  Field1 : Integer;
end record;
type A2 is new A1 with
null record;
```

B.
```
type B1 is tagged
record
  Field2 :  Integer;
end record;
type B2 is new B1 with
record
  Field2b :  Integer;
end record;
```

C.
```
type C1 is tagged
record
  Field3 : Integer;
end record;
type C2 is new C1 with
record
  Field3 : Integer;
end record;
```

D.
```
type D1 is tagged
record
  Field1 : Integer;
end record;
type D2 is new D1;
```

Explanations

A. Cannot extend a non-tagged type
B. Correct
C. Components must have distinct names
D. Types derived from a tagged type must have an extension

Lab

Tagged Derivation Lab

- Requirements
  - Create a type structure that could be used in a business
    - A **person** has some defining characteristics
    - An **employee** is a *person* with some employment information
    - A **staff member** is an *employee* with specific job information
  - Create primitive operations to read and print the objects
  - Create a main program to test the objects and operations
- Hints
  - Use **overriding** and **not overriding** as appropriate

Tagged Derivation Lab Solution - Types (Spec)

```
with Ada.Calendar;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Employee is
  type Person_T is tagged private;
  procedure Set_Name (O     : in out Person_T;
                      Value :        String);
  function Name (O : Person_T) return String;
  procedure Set_Birth_Date (O     : in out Person_T;
                            Value :        String);
  function Birth_Date (O : Person_T) return String;
  procedure Print (O : Person_T);

  type Employee_T is new Person_T with private;
  not overriding procedure Set_Start_Date (O     : in out Employee_T;
                                           Value :        String);
  not overriding function Start_Date (O : Employee_T) return String;
  overriding procedure Print (O : Employee_T);

  type Position_T is new Employee_T with private;
  not overriding procedure Set_Job (O     : in out Position_T;
                                    Value :        String);
  not overriding function Job (O : Position_T) return String;
  overriding procedure Print (O : Position_T);

private
  type Person_T is tagged record
    Name       : Unbounded_String;
    Birth_Date : Ada.Calendar.Time;
  end record;

  type Employee_T is new Person_T with record
    Employee_Id : Positive;
    Start_Date  : Ada.Calendar.Time;
  end record;

  type Position_T is new Employee_T with record
    Job : Unbounded_String;
  end record;
end Employee;
```

Tagged Derivation Lab Solution - Types (Body - Incomplete)

```
function To_String (T : Ada.Calendar.Time) return String is
begin
  return Month_Name (Ada.Calendar.Month (T)) &
   Integer'Image (Ada.Calendar.Day (T)) & "," &
   Integer'Image (Ada.Calendar.Year (T));
end To_String;

function From_String (S : String) return Ada.Calendar.Time is
  Date : constant String := S & " 12:00:00";
begin
  return Ada.Calendar.Formatting.Value (Date);
end From_String;

procedure Set_Name (O     : in out Person_T;
                   Value :        String) is
begin
  O.Name := To_Unbounded_String (Value);
end Set_Name;
function Name (O : Person_T) return String is (To_String (O.Name));

procedure Set_Birth_Date (O     : in out Person_T;
                          Value :        String) is
begin
  O.Birth_Date := From_String (Value);
end Set_Birth_Date;
function Birth_Date (O : Person_T) return String is (To_String (O.Birth_Date));
procedure Print (O : Person_T) is
begin
  Put_Line ("Name: " & Name (O));
  Put_Line ("Birthdate: " & Birth_Date (O));
end Print;

not overriding procedure Set_Start_Date (O     : in out Employee_T;
                                         Value :        String) is
begin
  O.Start_Date := From_String (Value);
end Set_Start_Date;
not overriding function Start_Date (O : Employee_T) return String is (To_String (O.Start_Date));
overriding procedure Print (O : Employee_T) is
begin
  Put_Line ("Name: " & Name (O));
  Put_Line ("Birthdate: " & Birth_Date (O));
  Put_Line ("Startdate: " & Start_Date (O));
end Print;
```

Tagged Derivation Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Employee;
procedure Main is
  function Read (Prompt : String) return String is
  begin
    Put (Prompt & "> ");
    return Get_Line;
  end Read;
  function Read_Date (Prompt : String) return String is (Read (Prompt & " (YYYY-MM-DD)"));

  Applicant : Employee.Person_T;
  Employ    : Employee.Employee_T;
  Staff     : Employee.Position_T;

begin
  Applicant.Set_Name (Read ("Applicant name"));
  Applicant.Set_Birth_Date (Read_Date ("   Birth Date"));

  Employ.Set_Name (Read ("Employee name"));
  Employ.Set_Birth_Date (Read_Date ("   Birth Date"));
  Employ.Set_Start_Date (Read_Date ("   Start Date"));

  Staff.Set_Name (Read ("Staff name"));
  Staff.Set_Birth_Date (Read_Date ("   Birth Date"));
  Staff.Set_Start_Date (Read_Date ("   Start Date"));
  Staff.Set_Job (Read ("   Job"));

  Applicant.Print;
  Employ.Print;
  Staff.Print;
end Main;
```

Summary

Summary

- Tagged derivation
  - Building block for OOP types in Ada
- Primitives rules for tagged types are trickier
  - Primitives **forbidden** below freeze point
  - **Unique** controlling parameter
  - Tip: Keep the number of tagged type per package low

Polymorphism

Introduction

Introduction

- **'Class** operator to categorize classes of types
- Type classes allow dispatching calls
    - Abstract types
    - Abstract subprograms
- Run-time call dispatch vs compile-time call dispatching

Classes of Types

Examples

```
package Class_Types is
   type Root_T is tagged null record;
   type Child1_T is new Root_T with null record;
   type Child2_T is new Root_T with null record;
   type Grandchild1_T is new Child1_T with null record;

   -- Root'Class = {Root_T, Child1_T, Child2_T, Grandchild1_T}
   -- Child1'Class = {Child1_T, Grandchild1_T} Child2'Class = {Child2_T}
   -- Granchild1'Class ={Grandchild1_T}
   procedure Test;

end Class_Types;

with Ada.Tags;    use Ada.Tags;
with Ada.Text_IO; use Ada.Text_IO;
package body Class_Types is
   Root_Object  : Root_T;
   Child_Object : Child1_T;

   Class_Object1 : Child1_T'Class := Child_Object;
   Class_Object2 : Root_T'Class   := Class_Object1;
   Class_Object3 : Root_T'Class   := Child_Object;
   -- Class_Object4 : Root_T'class; -- illegal

   procedure Do_Something (Object : in out Root_T'Class) is
   begin
      Put_Line
        ("Do_Something: " & Boolean'Image (Object in Root_T'Class) & " / " &
         Boolean'Image (Object in Child1_T'Class));
   end Do_Something;

   procedure Test is
   begin
      Put_Line (Boolean'Image (Class_Object1'Tag = Class_Object2'Tag));
      Put_Line (Boolean'Image (Class_Object2'Tag = Class_Object3'Tag));
      Do_Something (Root_Object);
      Do_Something (Child_Object);
      Do_Something (Class_Object1);
      Do_Something (Class_Object2);
      Do_Something (Class_Object3);
   end Test;
end Class_Types;

package Abstract_Types is
   type Root_T is abstract tagged record
      Field : Integer;
   end record;
   function Primitive1 (V : Root_T) return String is abstract;
   function Primitive2 (Prompt : String; V : Root_T) return String is
      (Prompt & "> " & Integer'Image (V.Field));

   type Child_T is abstract new Root_T with null record;
   -- Child_T does not need to redefine any primitives

   type Grandchild_T is new Child_T with null record;
   -- Grandchild_T is required to create a concrete version of Primitive2
   function Primitive1 (V : Grandchild_T) return String is
     (Integer'Image (V.Field));

   procedure Test;
end Abstract_Types;

with Ada.Text_IO; use Ada.Text_IO;
package body Abstract_Types is
   Object1 : constant Grandchild_T := (Field => 123);
   Object2 : constant Root_T'Class := Object1;

   procedure Test is
   begin
      Put_Line (Object1.Primitive1);
      Put_Line (Primitive2 ("Object1", Object2));
      Put_Line (Object2.Primitive1);
      Put_Line (Primitive2 ("Object2", Object2));
   end Test;

end Abstract_Types;

with Abstract_Types;
with Class_Types;
procedure Test is
begin
   Class_Types.Test;
   Abstract_Types.Test;
end Test;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/180_polymorphism.html#classes-of-types

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**

```
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**

Indefinite type

- A class wide type is an indefinite type
  - Just like an unconstrained array or a record with a discriminant
- Properties and constraints of indefinite types apply
  - Can be used for parameter declarations
  - Can be used for variable declaration with initialization

```
procedure Main is
   type T is tagged null record;
   type D is new T with null record;
   procedure P (X : in out T'Class) is null;
   Obj : D;
   Dc  : D'Class := Obj;
   Tc1 : T'Class := Dc;
   Tc2 : T'Class := Obj;
   -- initialization required in class-wide declaration
   Tc3 : T'Class; -- compile error
   Dc2 : D'Class; -- compile error
begin
   P (Dc);
   P (Obj);
end Main;
```

Testing the type of an object

- The tag of an object denotes its type
- It can be accessed through the **'Tag** attribute
- Applies to both objects and types
- Membership operator is available to check the type against a hierarchy

```
type Parent is tagged null record;
type Child is new Parent with null record;
Parent_Obj : Parent; -- Parent_Obj'Tag = Parent'Tag
Child_Obj  : Child;  -- Child_Obj'Tag = Child'Tag
Parent_Class_1 : Parent'Class := Parent_Obj;
                 -- Parent_Class_1'Tag = Parent'Tag
Parent_Class_2 : Parent'Class := Child_Obj;
                 -- Parent_Class_2'Tag = Child'Tag
Child_Class    : Child'Class := Child(Parent_Class_2);
                 -- Child_Class'Tag  = Child'Tag

B1 : Boolean := Parent_Class_1 in Parent'Class;       -- True
B2 : Boolean := Parent_Class_1'Tag = Child'Class'Tag; -- False
B3 : Boolean := Child_Class'Tag = Parent'Class'Tag;   -- False
B4 : Boolean := Child_Class in Child'Class;           -- True
```

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

Abstract Types Ada vs C++

- Ada

```
type Root is abstract tagged record
   F : Integer;
end record;
procedure P1 (V : Root) is abstract;
procedure P2 (V : Root);
type Child is abstract new Root with null record;
type Grand_Child is new Child with null record;

overriding  -- Ada 2005 and later
procedure P1 (V : Grand_Child);
```

- C++

```
class Root {
   public:
      int F;
      virtual void P1 (void) = 0;
      virtual void P2 (void);
};
class Child : public Root {
};
class Grand_Child {
   public:
      virtual void P1 (void);
};
```

Relation to Primitives

Ada 2012

- Warning: Subprograms with parameter of type **T'Class** are primitives of **T'Class**, not **T**

```
type Root is null record;
procedure P (V : Root'Class);
type Child is new Root with null record;
-- This does not override P!
overriding procedure P (V : Child'Class);
```

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

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

Dispatching and Redispatching

Examples

```
package Types is

   type Root_T is tagged null record;
   function Primitive (V : Root_T) return String is ("Root_T");

   type Child_T is new Root_T with null record;
   function Primitive (V : Child_T) return String is ("Child_T");

end Types;

with Ada.Text_IO; use Ada.Text_IO;
with Types;       use Types;
procedure Test_Dispatching_And_Redispatching is

   Root_Object  : Root_T;
   Child_Object : Child_T;

   V1 : constant Root_T'Class  := Root_Object;
   V2 : constant Root_T'Class  := Child_Object;
   V3 : constant Child_T'Class := Child_Object;

begin

   Put_Line (Primitive (V1));
   Put_Line (Primitive (V2));
   Put_Line (Primitive (V3));

end Test_Dispatching_And_Redispatching;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/180_polymorphism.html#dispatching-and-redispatching

Calls on class-wide types (1/3)

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

```
type Root is 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 (2/3)

- 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 ();
```

Calls on class-wide types (3/3)

- It is still possible to force a call to be static using a conversion of view

Ada

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

C++

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

Definite and class wide views

- In C++, dispatching occurs only on pointers
- In Ada, dispatching occurs only on class wide views

```
type Root is tagged null record;
procedure P1 (V : Root);
procedure P2 (V : Root);
type Child is new Root with null record;
overriding procedure P2 (V : Child);
procedure P1 (V : Root) is
begin
   P2 (V); -- always calls P2 from Root
end P1;
procedure Main is
   V1 : Root'Class :=
        Child'(others => <>);
begin
   -- Calls P1 from the implicitly overridden subprogram
   -- Calls P2 from Root!
   V1.P1;
```

Redispatching

- **tagged** types are always passed by reference
  - The original object is not copied
- Therefore, it is possible to convert them to different views

```
type Root is tagged null record;
procedure P1 (V : Root);
procedure P2 (V : Root);
type Child is new Root with null record;
overriding procedure P2 (V : Child);
```

Redispatching Example

```
procedure P1 (V : Root) is
   V_Class : Root'Class renames
             Root'Class (V); -- naming of a view
begin
   P2 (V);                 -- static: uses the definite view
   P2 (Root'Class (V));    -- dynamic: (redispatching)
   P2 (V_Class);           -- dynamic: (redispatching)

   -- Ada 2005 "distinguished receiver" syntax
   V.P2;                   -- static: uses the definite view
   Root'Class (V).P2;      -- dynamic: (redispatching)
   V_Class.P2;             -- dynamic: (redispatching)
end P1;
```

Quiz

```
package P is
   type Root is tagged null record;
   function F1 (V : Root) return Integer is (101);
   type Child is new Root with null record;
   function F1 (V : Child) return Integer is (201);
   type Grandchild is new Child with null record;
   function F1 (V : Grandchild) return Integer is (301);
end P;

with P1; use P1;
procedure Main is
   Z : Root'Class := Grandchild'(others => <>);
```

What is the value returned by `F1 (Child'Class (Z));`?

A. 301
B. 201
C. 101
D. Compilation error

Quiz

```
package P is
   type Root is tagged null record;
   function F1 (V : Root) return Integer is (101);
   type Child is new Root with null record;
   function F1 (V : Child) return Integer is (201);
   type Grandchild is new Child with null record;
   function F1 (V : Grandchild) return Integer is (301);
end P;

with P1; use P1;
procedure Main is
   Z : Root'Class := Grandchild'(others => <>);
```

What is the value returned by `F1 (Child'Class (Z));`?

A. ***301***
B. 201
C. 101
D. Compilation error

Explanations

A. Correct
B. Would be correct if the cast was `Child` - `Child'Class` leaves the object as `Grandchild`
C. Object is initialized to something in `Root'class`, but it doesn't have to be `Root`
D. Would be correct if function parameter types were `'Class`

Exotic Dispatching Operations

Examples

```ada
package Types is
   type Root_T is tagged record
      Field : Integer;
   end record;
   function Primitive (Left  : Root_T; Right : Root_T) return Integer is
      (Left.Field + Right.Field);
   function "=" (Left  : Root_T; Right : Root_T) return Boolean is
      (Left.Field in Right.Field - 1 .. Right.Field + 1);
   function Constructor (I : Integer := 0) return Root_T is ((Field => I));

   type Child_T is new Root_T with null record;
   overriding function Primitive (Left  : Child_T; Right : Child_T) return Integer is
      (Left.Field * Right.Field);
   overriding function "=" (Left  : Child_T; Right : Child_T) return Boolean is
      (Right.Field in Left.Field - 1 .. Left.Field + 1);
   -- function Constructor ( I : Integer := 0 ) return child_T; -- inherited from Root_t

   type Child2_T is new Root_T with record
      Field2 : Integer;
   end record;
   overriding function Primitive (Left  : Child2_T; Right : Child2_T) return Integer is
      (Left.Field * Right.Field);
   overriding function "=" (Left  : Child2_T; Right : Child2_T) return Boolean is
      (Left.Field = Right.Field);
   -- must create a constructor because new fields added
   function Constructor (I : Integer := 0) return Child2_T is ((I, I));
end Types;

with Ada.Text_IO; use Ada.Text_IO;
with Types;       use Types;
procedure Test_Exotic_Dispatching_Operations is
   R1  : constant Root_T       := (Field => 10);
   R2  : constant Root_T       := (Field => 20);
   C1  : constant Child_T      := (Field => 10);
   Cl1 : constant Root_T'Class := R1;
   Cl2 : constant Root_T'Class := R2;
   Cl3 : constant Root_T'Class := C1;

   procedure Test_Primitive is
   begin
      Put_Line ("Primitive");
      Put_Line (Integer'Image (Primitive (R1, R2)));                 -- static:  ok
      -- Put_Line (Integer'Image (Primitive (R1, C1)));              -- static: error
      Put_Line (Integer'Image (Primitive (Cl1, Cl2)));               -- dynamic: ok
      -- Put_Line (Integer'Image (Primitive (R1, Cl1)));             -- static: error
      Put_Line (Integer'Image (Primitive (Root_T'Class (R1), Cl1))); -- dynamic: ok
      Put_Line (Integer'Image (Primitive (Cl1, Cl3)));               -- dynamic: error
   end Test_Primitive;

   procedure Test_Equality is
   begin
      Put_Line ("Equality");
      Put_Line ("Cl1 = Cl2 " & Boolean'Image (Cl1 = Cl2));
      Put_Line ("Cl2 = Cl3 " & Boolean'Image (Cl2 = Cl3));
      Put_Line ("Cl3 = Cl1 " & Boolean'Image (Cl3 = Cl1));
   end Test_Equality;

   procedure Test_Constructor is
      -- Static call to Root_T primitive
      V1 : Root_T'Class := Root_T'(Constructor);
      V2 : Root_T'Class := V1;
      -- Static call to Child2_T primitive
      V3 : Root_T'Class := Child2_T'(Constructor);
      -- V4 : Root_T'Class := Constructor; -- What is the tag of V4?
   begin
      -- No
      -- V1 := Constructor;

      -- Yes
      V1 := Root_T'(Constructor);
   end Test_Constructor;

begin
   Test_Equality;
   Test_Constructor;
   Test_Primitive;
end Test_Exotic_Dispatching_Operations;
```

https://learn.adacore.com/training_examples/fundamentals_of_ada/180_polymorphism.html#exotic-dispatching-operations

Multiple dispatching operands

- Primitives with multiple dispatching operands are allowed if all operands are of the same type

```
type Root is null tagged record;
procedure P (Left : Root; Right : Root);
type Child is new Root with null record;
overriding procedure P (Left : Child; Right : Child);
```

- At call time, all actual parameters' tags have to match, either statically or dynamically

```
R1, R2 : Root;
C1, C2 : Child;
Cl1 : Root'Class := R1;
Cl2 : Root'Class := R2;
Cl3 : Root'Class := C1;
...
P (R1, R2);              -- static:  ok
P (R1, C1);              -- static:  error
P (Cl1, Cl2);            -- dynamic: ok
P (Cl1, Cl3);            -- dynamic: error
P (R1, Cl1);             -- static:  error
P (Root'Class (R1), Cl1); -- dynamic: ok
```

Special case for equality

- Overriding the default equality for a **tagged** type involves the use of a function with multiple controlling operands
- As in general case, static types of operands have to be the same
- If dynamic types differ, equality returns false instead of raising exception

```
type Root is null tagged record;
function "=" (L : Root; R : Root) return Boolean;
type Child is new Root with null record;
overriding function "=" (L : Child; R : Child) return Boolean;
R1, R2 : Root;
C1, C2 : Child;
Cl1 : Root'Class := R1;
Cl2 : Root'Class := R2;
Cl3 : Root'Class := C1;
...
-- overridden "=" called via dispatching
if Cl1 = Cl2 then [...]
if Cl1 = Cl3 then [...] -- returns false
```

Controlling result (1/2)

- The controlling operand may be the return type
  - This is known as the constructor pattern

```
type Root is tagged null record;
function F (V : Integer) return Root;
```

- If the child adds fields, all such subprograms have to be overridden

```
type Root is tagged null record;
function F (V : Integer) return Root;

type Child is new Root with null record;
-- OK, F is implicitly inherited

type Child1 is new Root with record
   X : Integer;
end record;
-- ERROR no implicitly inherited function F
```

- Primitives returning abstract types have to be abstract

```
type Root is abstract tagged null record;
function F (V : Integer) return Root is abstract;
```

Controlling result (2/2)

- Primitives returning **tagged** types can be used in a static context

```
type Root is tagged null record;
function F return Root;
type Child is new Root with null record;
function F return Child;
V : Root := F;
```

- In a dynamic context, the type has to be known to correctly dispatch

```
V1 : Root'Class := Root'(F);  -- Static call to Root primitive
V2 : Root'Class := V1;
V3 : Root'Class := Child'(F); -- Static call to Child primitive
V4 : Root'Class := F;         -- What is the tag of V4?
...
V1 := F; -- Dispatching call to Root primitive
V2 := F; -- Dispatching call to Root primitive
V3 := F; -- Dispatching call to Child primitive
```

- No dispatching is possible when returning access types

Lab

Polymorphism Lab

- Requirements
  - Create a multi-level types hierarchy of shapes
    - Level 1: Shape $\rightarrow$ Quadrilateral | Triangle
    - Level 2: Quadrilateral $\rightarrow$ Square
  - Types should have the following primitive operations
    - Description
    - Number of sides
    - Perimeter
  - Create a main program to print information about multiple shapes
    - Create a nested subprogram that takes a shape and prints all relevant information
- Hints
  - Top-level type should be abstract
    - But can have concrete operations
  - Nested subprogram in **main** should take a shape class parameter

Polymorphism Lab Solution - Shapes (Spec)

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package Shapes is
  type Float_T is digits 6;
  type Vertex_T is record
    X : Float_T;
    Y : Float_T;
  end record;
  type Vertices_T is array (Positive range <>) of Vertex_T;

  type Shape_T is abstract tagged record
    Description : Unbounded_String;
  end record;
  function Get_Description (Shape : Shape_T'Class) return String;
  function Number_Of_Sides (Shape : Shape_T) return Natural is abstract;
  function Perimeter (Shape : Shape_T) return Float_T is abstract;

  type Quadrilateral_T is new Shape_T with record
    Sides : Vertices_T (1 .. 4);
  end record;
  function Number_Of_Sides (Shape : Quadrilateral_T) return Natural;
  function Perimeter (Shape : Quadrilateral_T) return Float_T;

  type Square_T is new Quadrilateral_T with null record;
  function Perimeter (Shape : Square_T) return Float_T;

  type Triangle_T is new Shape_T with record
    Sides : Vertices_T (1 .. 3);
  end record;
  function Number_Of_Sides (Shape : Triangle_T) return Natural;
  function Perimeter (Shape : Triangle_T) return Float_T;
end Shapes;
```

Polymorphism Lab Solution - Shapes (Body)

```
with Ada.Numerics.Generic_Elementary_Functions;
package body Shapes is
  package Math is new Ada.Numerics.Generic_Elementary_Functions (Float_T);

  function Distance (Vertex1 : Vertex_T;
                     Vertex2 : Vertex_T)
                     return Float_T is
   (Math.Sqrt ((Vertex1.X - Vertex2.X)**2 + (Vertex1.Y - Vertex2.Y)**2));

  function Perimeter (Vertices : Vertices_T) return Float_T is
    Ret_Val : Float_T := 0.0;
  begin
    for I in Vertices'First .. Vertices'Last - 1 loop
      Ret_Val := Ret_Val + Distance (Vertices (I), Vertices (I + 1));
    end loop;
    Ret_Val := Ret_Val + Distance (Vertices (Vertices'Last), Vertices (Vertices'First));
    return Ret_Val;
  end Perimeter;

  function Get_Description (Shape : Shape_T'Class) return String is (To_String (Shape.Description));

  function Number_Of_Sides (Shape : Quadrilateral_T) return Natural is (4);
  function Perimeter (Shape : Quadrilateral_T) return Float_T is (Perimeter (Shape.Sides));

  function Perimeter (Shape : Square_T) return Float_T is (4.0 * Distance (Shape.Sides (1), Shape.Sides (2)));

  function Number_Of_Sides (Shape : Triangle_T) return Natural is (3);
  function Perimeter (Shape : Triangle_T) return Float_T is (Perimeter (Shape.Sides));
end Shapes;
```

Polymorphism Lab Solution - Main

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;
with Shapes;                use Shapes;
procedure Main is

  Rectangle : constant Shapes.Quadrilateral_T :=
   (Description => To_Unbounded_String ("rectangle"),
    Sides       => ((0.0, 10.0), (0.0, 20.0), (1.0, 20.0), (1.0, 10.0)));
  Triangle : constant Shapes.Triangle_T :=
   (Description => To_Unbounded_String ("triangle"),
    Sides       => ((0.0, 0.0), (0.0, 3.0), (4.0, 0.0)));
  Square : constant Shapes.Square_T :=
   (Description => To_Unbounded_String ("square"),
    Sides       => ((0.0, 1.0), (0.0, 2.0), (1.0, 2.0), (1.0, 1.0)));

  procedure Describe (Shape : Shapes.Shape_T'Class) is
  begin
    Put_Line (Shape.Get_Description);
    if Shape not in Shapes.Shape_T then
      Put_Line ("  Number of sides:" & Integer'Image (Shape.Number_Of_Sides));
      Put_Line ("  Perimeter:" & Shapes.Float_T'Image (Shape.Perimeter));
    end if;
  end Describe;
begin

  Describe (Rectangle);
  Describe (Triangle);
  Describe (Square);
end Main;
```

Summary

Summary

- **'Class** operator
    - Allows subprograms to be used for multiple versions of a type
- Dispatching
    - Abstract types require concrete versions
    - Abstract subprograms allow template definitions
        - Need an implementation for each abstract type referenced
- Run-time call dispatch vs compile-time call dispatching
    - Compiler resolves appropriate call where it can
    - Run-time resolves appropriate call where it can
    - If not resolved, exception

Multiple Inheritance

Introduction

Multiple Inheritance Is Forbidden In Ada

- There are potential conflicts with multiple inheritance
- Some languages allow it: ambiguities have to be resolved when entities are referenced
- Ada forbids it to improve integration

```
type Graphic is tagged record
   X, Y : Float;
end record;
function Get_X (V : Graphic) return Float;

type Shape is tagged record
   X, Y : Float;
end record;
function Get_X (V : Shape) return Float;

type Displayable_Shape is new Shape and Graphic with ...
```

Multiple Inheritance - Safe Case

- If only one type has concrete operations and fields, this is fine

```
type Graphic is abstract tagged null record;
function Get_X (V : Graphic) return Float is abstract;

type Shape is tagged record
   X, Y : Float;
end record;
function Get_X (V : Shape) return Float;

type Displayable_Shape is new Shape and Graphic with ...
```

- This is the definition of an interface (as in Java)

```
type Graphic is interface;
function Get_X (V : Graphic) return Float is abstract;

type Shape is tagged record
   X, Y : Float;
end record;
function Get_X (V : Shape) return Float;

type Displayable_Shape is new Shape and Graphic with ...
```

Interfaces

Interfaces - Rules

- An interface is a tagged type marked interface, containing
  - Abstract primitives
  - Null primitives
  - No fields
- Null subprograms provide default empty bodies to primitives that can be overridden

```
type I is interface;
procedure P1 (V : I) is abstract;
procedure P2 (V : access I) is abstract
function F return I is abstract;
procedure P3 (V : I) is null;
```

- Note: null can be applied to any procedure (not only used for interfaces)

Interface Derivation

- An interface can be derived from another interface, adding primitives

```
type I1 is interface;
procedure P1 (V : I) is abstract;
type I2 is interface and I1;
Procedure P2 (V : I) is abstract;
```

- A tagged type can derive from several interfaces and can derive from one interface several times

```
type I1 is interface;
type I2 is interface and I1;
type I3 is interface;

type R is new I1 and I2 and I3 ...
```

- A tagged type can derive from a single tagged type and several interfaces

```
type I1 is interface;
type I2 is interface and I1;
type R1 is tagged null record;

type R2 is new R1 and I1 and I2 ...
```

Interfaces And Privacy

- If the partial view of the type is tagged, then both the partial and the full view must expose the same interfaces

```
package Types is

   type I1 is interface;
   type R is new I1 with private;

private

   type R is new I1 with record ...
```

Limited Tagged Types And Interfaces

- When a tagged type is limited in the hierarchy, the whole hierarchy has to be limited
- Conversions to interfaces are "just conversions to a view"
  - A view may have more constraints than the actual object
- `limited` interfaces can be implemented by BOTH limited types and non-limited types
- Non-limited interfaces have to be implemented by non-limited types

Lab

Multiple Inheritance Lab

- Requirements
  - Create a tagged type to define shapes
    - Possible components could include location of shape
  - Create an interface to draw lines
    - Possible accessor functions could include line color and width
  - Create a new type inheriting from both of the above for a "printable object"
    - Implement a way to print the object using **Ada.Text_IO**
    - Does not have to be fancy!
  - Create a "printable object" type to draw something (rectangle, triangle, etc)
- Hints
  - This example is taken from Barnes' *Programming in Ada 2012* Section 21.2

Inheritance Lab Solution - Data Types

```
package Base_Types is
   type Coordinate_T is record
      X_Coord : Integer;
      Y_Coord : Integer;
   end record;

   type Line_T is array (1 .. 2) of Coordinate_T;
   -- convert Line_T so lowest X value is first
   function Ordered (Line : Line_T) return Line_T;
   type Lines_T is array (Natural range <>) of Line_T;

   type Color_Range_T is mod 255;
   type Color_T is record
      Red   : Color_Range_T;
      Green : Color_Range_T;
      Blue  : Color_Range_T;
   end record;

private
   function Ordered (Line : Line_T) return Line_T is
     (if Line (1).X_Coord > Line (2).X_Coord then (Line (2), Line (1)) else Line);
end Base_Types;
```

Inheritance Lab Solution - Shapes

```
with Base_Types;
package Geometry is
   type Object_T is abstract tagged private;

private
   type Object_T is abstract tagged record
      Origin : Base_Types.Coordinate_T;
   end record;
   function Origin (Object : Object_T'Class)
                    return Base_Types.Coordinate_T is
      (Object.Origin);
end Geometry;
```

Inheritance Lab Solution - Drawing (Spec)

```
with Base_Types;
package Line_Draw is
   type Object_T is interface;
   procedure Set_Color (Object : in out Object_T;
                        Color  :        Base_Types.Color_T)
                        is abstract;
   function Color (Object : Object_T)
                   return Base_Types.Color_T
                   is abstract;
   procedure Set_Pen (Object : in out Object_T;
                      Size   :        Positive)
                      is abstract;
   function Pen (Object : Object_T)
                 return Positive
                 is abstract;
   function Convert (Object : Object_T)
                     return Base_Types.Lines_T
                     is abstract;
   procedure Print (Object : Object_T'Class);
end Line_Draw;
```

Inheritance Lab Solution - Drawing (Body)

```
with Ada.Text_IO;
with Line_Draw.Graph;
package body Line_Draw is
   procedure Fill_Matrix (Matrix : in out Graph.Matrix_T;
                          Line   : in     Base_Types.Line_T) is
      M, B     : Float;
      Vertical : Boolean;
   begin
      Graph.Find_Slope_And_Intercept (Line, M, B, Vertical);
      if Vertical then
         for Y in Integer'Min (Line (1).Y_Coord, Line (2).Y_Coord) ..
                  Integer'Max (Line (1).Y_Coord, Line (2).Y_Coord) loop
            Matrix (Line (1).X_Coord, Y) := 'X';
         end loop;
      elsif Graph.Rise (Line) > Graph.Run (Line) then
         Graph.Fill_Matrix_Vary_Y (Matrix, Line, M, B);
      else
         Graph.Fill_Matrix_Vary_X (Matrix, Line, M, B);
      end if;
   end Fill_Matrix;

   procedure Print (Object : Object_T'Class) is
      Lines        : constant Base_Types.Lines_T := Object.Convert;
      Max_X, Max_Y : Integer                     := Integer'First;
      Min_X, Min_Y : Integer                     := Integer'Last;
   begin
      for Line of Lines loop
         for Coord of Line loop
            Max_X := Integer'Max (Max_X, Coord.X_Coord);
            Min_X := Integer'Min (Min_X, Coord.X_Coord);
            Max_Y := Integer'Max (Max_Y, Coord.Y_Coord);
            Min_Y := Integer'Min (Min_Y, Coord.Y_Coord);
         end loop;
      end loop;
      declare
         Matrix : Graph.Matrix_T (Min_X .. Max_X, Min_Y .. Max_Y) := (others => (others => ' '));
      begin
         for Line of Lines loop
            Fill_Matrix (Matrix, Base_Types.Ordered (Line));
         end loop;
         for Y in Matrix'Range (2) loop
            for X in Matrix'Range (1) loop
               Ada.Text_IO.Put (Matrix (X, Y));
            end loop;
            Ada.Text_IO.New_Line;
         end loop;
      end;
   end Print;
end Line_Draw;
```

Inheritance Lab Solution - Graphics (Spec)

```
package Line_Draw.Graph is
   type Matrix_T is array (Integer range <>, Integer range <>) of Character;

   procedure Find_Slope_And_Intercept
     (Line     : in     Base_Types.Line_T;
      M        :    out Float;
      B        :    out Float;
      Vertical :    out Boolean);

   function Rise (Line : Base_Types.Line_T) return Float;
   function Run (Line : Base_Types.Line_T) return Float;

   procedure Fill_Matrix_Vary_X
     (Matrix : in out Matrix_T;
      Line   : in     Base_Types.Line_T;
      M      : in     Float;
      B      : in     Float);
   procedure Fill_Matrix_Vary_Y
     (Matrix : in out Matrix_T;
      Line   : in     Base_Types.Line_T;
      M      : in     Float;
      B      : in     Float);
end Line_Draw.Graph;
```

Inheritance Lab Solution - Graphics (Body)

```
package body Line_Draw.Graph is
   function Rise (Line : Base_Types.Line_T) return Float is
      (Float (Line (2).Y_Coord - Line (1).Y_Coord));
   function Run (Line : Base_Types.Line_T) return Float is
      (Float (Line (2).X_Coord - Line (1).X_Coord));

   procedure Fill_Matrix_Vary_Y (Matrix : in out Matrix_T;
                                 Line   : in     Base_Types.Line_T;
                                 M      : in     Float;
                                 B      : in     Float) is
                                 X : Integer;
   begin
      for Y in Line (1).Y_Coord .. Line (2).Y_Coord loop
         X             := Integer ((Float (Y) - B) / M);
         Matrix (X, Y) := 'X';
      end loop;
   end Fill_Matrix_Vary_Y;

   procedure Fill_Matrix_Vary_X (Matrix : in out Matrix_T;
                                 Line   : in     Base_Types.Line_T;
                                 M      : in     Float;
                                 B      : in     Float) is
      Y : Integer;
   begin
      for X in Line (1).X_Coord .. Line (2).X_Coord loop
         Y             := Integer (M * Float (X) + B);
         Matrix (X, Y) := 'X';
      end loop;
   end Fill_Matrix_Vary_X;

   procedure Find_Slope_And_Intercept (Line     : in     Base_Types.Line_T;
                                       M        :    out Float;
                                       B        :    out Float;
                                       Vertical :    out Boolean) is
   begin
      if Run (Line) = 0.0 then
         M        := 0.0;
         B        := 0.0;
         Vertical := True;
      else
         M        := Rise (Line) / Run (Line);
         B        := Float (Line (1).Y_Coord) - M * Float (Line (1).X_Coord);
         Vertical := False;
      end if;
   end Find_Slope_And_Intercept;

end Line_Draw.Graph;
```

Inheritance Lab Solution - Printable Object

```
with Geometry;
with Line_Draw;
with Base_Types;
package Printable_Object is
   type Object_T is abstract new Geometry.Object_T and Line_Draw.Object_T with private;
   procedure Set_Color (Object : in out Object_T;
                        Color  :        Base_Types.Color_T);
   function Color (Object : Object_T) return Base_Types.Color_T;
   procedure Set_Pen (Object : in out Object_T;
                      Size   :        Positive);
   function Pen (Object : Object_T) return Positive;
private
   type Object_T is abstract new Geometry.Object_T and Line_Draw.Object_T with
   record
      Color    : Base_Types.Color_T := (0, 0, 0);
      Pen_Size : Positive           := 1;
   end record;
end Printable_Object;

package body Printable_Object is

   procedure Set_Color (Object : in out Object_T;
                        Color  :        Base_Types.Color_T) is
   begin
      Object.Color := Color;
   end Set_Color;
   function Color (Object : Object_T) return Base_Types.Color_T is (Object.Color);

   procedure Set_Pen (Object : in out Object_T;
                      Size   :        Positive) is
   begin
      Object.Pen_Size := Size;
   end Set_Pen;
   function Pen (Object : Object_T) return Positive is (Object.Pen_Size);
end Printable_Object;
```

Inheritance Lab Solution - Rectangle

```
with Base_Types;
with Printable_Object;
package Rectangle is
   subtype Lines_T is Base_Types.Lines_T (1 .. 4);
   type Object_T is new Printable_Object.Object_T with private;
   procedure Set_Lines (Object : in out Object_T;
                        Lines  :        Lines_T);
   function Lines (Object : Object_T) return Lines_T;
private
   type Object_T is new Printable_Object.Object_T with record
      Lines : Lines_T;
   end record;
   function Convert (Object : Object_T) return Base_Types.Lines_T is
      (Object.Lines);
end Rectangle;

package body Rectangle is
   procedure Set_Lines (Object : in out Object_T;
                        Lines  :        Lines_T) is
   begin
      Object.Lines := Lines;
   end Set_Lines;

   function Lines (Object : Object_T) return Lines_T is (Object.Lines);
end Rectangle;
```

Inheritance Lab Solution - Main

```
with Base_Types;
with Rectangle;
procedure Main is

   Object : Rectangle.Object_T;
   Line1  : constant Base_Types.Line_T := ((1, 1), (1, 10));
   Line2  : constant Base_Types.Line_T := ((6, 6), (6, 15));
   Line3  : constant Base_Types.Line_T := ((1, 1), (6, 6));
   Line4  : constant Base_Types.Line_T := ((1, 10), (6, 15))
begin
   Object.Set_Lines ((Line1, Line2, Line3, Line4));
   Object.Print;
end Main;
```

Summary

Summary

- Interfaces must be used for multiple inheritance
  - Usually combined with `tagged` types, but not necessary
  - By using only interfaces, only accessors are allowed
- Typically there are other ways to do the same thing
  - In our example, the conversion routine could be common to simplify things
- But interfaces force the compiler to determine when operations are missing

Advanced Exceptions

Introduction

Advanced Usages

- Language-defined exceptions raising cases
- Re-raising
- Raising and handling from elaboration
- Manipulating an exception with identity
    - Re-raising
    - Copying

Handlers

Exceptions Raised In Exception Handlers

- Go immediately to caller unless also handled
- Goes to caller in any case, as usual

```
begin
  ...
exception
  when Some_Error =>
    declare
      New_Data : Some_Type;
    begin
      P( New_Data );
      ...
    exception
      when ...
    end;
end;
```

Language-Defined Exceptions

Constraint_Error

- Caused by violations of constraints on range, index, etc.
- The most common exceptions encountered

```
K : Integer range 1 .. 10;
...
K := -1;
```

```
L : array (1 .. 100) of Some_Type;
...
L (400) := SomeValue;
```

Program_Error

- When runtime control structure is violated
  - Elaboration order errors and function bodies
- When implementation detects bounded errors
  - Discussed momentarily

```
function F return Some_Type is
begin
  if something then
    return Some_Value;
  end if; -- program error - no return statement
end F;
```

Storage_Error

- When insufficient storage is available
- Potential causes
  - Declarations
  - Explicit allocations
  - Implicit allocations

```
Data : array (1..1e20) of Big_Type;
```

Explicitly-Raised Exceptions

- Raised by application via `raise` statements
  - Named exception becomes active
- Syntax

```
raise_statement ::= raise; |
    raise exception_name
    [with string_expression];
```

  - `with string_expression` only available in Ada 2005 and later
- A `raise` by itself is only allowed in handlers (more later)

```
if Unknown (User_ID) then
  raise Invalid_User;
end if;

if Unknown (User_ID) then
  raise Invalid_User
     with "Attempt by " &
          Image (User_ID);
end if;
```

Propagation

Partially Handling Exceptions

- Handler eventually re-raises the current exception
- Achieved using **raise** by itself, since re-raising
  - Current active exception is then propagated to caller

```
procedure Joy_Ride is
   ...
begin
  while not Bored loop
    Steer_Aimlessly (Bored);
    Consume_Fuel (Hot_Rod);
  end loop;
exception
  when Fuel_Exhausted =>
    Pull_Over;
    raise; -- no gas available
end Joy_Ride;
```

Typical Partial Handling Example

- Log (or display) the error and re-raise to caller
    - Same exception or another one

```
procedure Get (Item : out Integer;   From : in File) is
begin
  Ada.Integer_Text_IO.Get (From, Item);
exception
  when Ada.Text_IO.End_Error =>
    Display_Error ("Attempted read past end of file");
    raise Error;
  when Ada.Text_IO.Mode_Error =>
    Display_Error ("Read from file opened for writing");
    raise Error;
  when Ada.Text_IO.Status_Error =>
    Display_Error ("File must be opened prior to use");
    raise Error;
  when others =>
    Display_Error ( "Error in Get(Integer) from file" );
    raise;
end Get;
```

Exceptions Raised During Elaboration

- I.e., those occurring before the `begin`
- Go immediately to the caller
- No handlers in that frame are applicable
    - Could reference declarations that failed to elaborate!

```
procedure P (Output : out BigType) is
  -- storage error handled by caller
  N : array (Positive) of BigType;
  ...
begin
  ...
exception
  when Storage_Error =>
    -- failure to define N not handled here
    Output := N (1); -- if it was, this wouldn't work
    ...
end P;
```

Handling Elaboration Exceptions

```
procedure Test is
  procedure P is
    X : Positive := 0;  -- Constraint_Error!
  begin
    ...
  exception
    when Constraint_Error =>
      Ada.Text_IO.Put_Line ("Got it in P");
  end P;
begin
  P;
exception
  when Constraint_Error =>
    Ada.Text_IO.Put_Line ("Got Constraint_Error in Test");
end Test;
```

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Exception_Test (Input_Value : Integer) is
   Known_Problem : exception;
   function F (P : Integer) return Integer is
   begin
      if P > 0 then
         return P * P;
      end if;
   exception
      when others => raise Known_Problem;
   end F;
   procedure P (X : Integer) is
      A : array (1 .. F (X)) of Float;
   begin
      A := (others => 0.0);
   exception
      when others => raise Known_Problem;
   end P;
begin
   P ( Input_Value );
   Put_Line ( "Success" );
exception
   when Known_Problem => Put_Line ("Known problem");
   when others => Put_Line ("Unknown problem");
end Exception_Test;
```

What will get printed for these values of Input_Value?

A. Integer'Last
B. Integer'First
C. 10000
D. 100

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Exception_Test (Input_Value : Integer) is
   Known_Problem : exception;
   function F (P : Integer) return Integer is
   begin
      if P > 0 then
         return P * P;
      end if;
   exception
      when others => raise Known_Problem;
   end F;
   procedure P (X : Integer) is
      A : array (1 .. F (X)) of Float;
   begin
      A := (others => 0.0);
   exception
      when others => raise Known_Problem;
   end P;
begin
   P ( Input_Value );
   Put_Line ( "Success" );
exception
   when Known_Problem => Put_Line ("Known problem");
   when others => Put_Line ("Unknown problem");
end Exception_Test;
```

What will get printed for these values of Input_Value?

A.	Integer'Last	Known Problem
B.	Integer'First	Unknown Problem
C.	10000	Unknown Problem
D.	100	Success

Explanations

A → When `F` is called with a large `P`, its own exception handler captures the exception and raises `Constraint_Error` (which the main exception handler processes)

B/C → When the creation of `A` fails (due to `Program_Error` from passing `F` a negative number or `Storage_Error` from passing `F` a large number), then `P` raises an exception during elaboration, which is propagated to `Main`

Exceptions as Objects

Exceptions Scope

- Some differences for scope and visibility
    - May be propagated out of scope
    - Hidden predefined exceptions are still available
    - Are not dynamically allocated (unlike variables)
        - A rarely-encountered issue involving recursion

Example Propagation Beyond Scope

```
package P is
  procedure Q;
end P;
package body P is
  Error : exception;
  procedure Q is
  begin
    ...
    raise Error;
  end Q;
end P;
```

```
with P;
procedure Client is
begin
  P.Q;
exception
  -- not visible
  when P.Error =>
     ...
  -- captured here
  when others =>
     ...
end Client;
```

User Subprogram Parameter Example

```
with Ada.Exceptions; use Ada.Exceptions;
procedure Display_Exception
    (Error : in Exception_Occurrence)
is
  Msg : constant String := Exception_Message (Error);
  Info : constant String := Exception_Information (Error);
begin
  New_Line;
  if Info /= "" then
    Put ("Exception information => ");
    Put_Line (Info);
  elsif Msg /= "" then
    Put ("Exception message => ");
    Put_Line (Msg);
  else
    Put ("Exception name => ");
    Put_Line (Exception_Name (Error));
  end if;
end Display_Exception;
```

Exception Identity

- Attribute 'Identity converts exceptions to the type

```
package Ada.Exceptions is
   ...
   type Exception_Id is private;
   ...
   procedure Raise_Exception( E : in Exception_Id;
                              Message : in String := "" );
   ...
end Ada.Exceptions;
```

- Primary use is raising exceptions procedurally

```
Foo : exception;
...
Ada.Exceptions.Raise_Exception (Foo'Identity,
                                Message => "FUBAR!");
```

Re-Raising Exceptions Procedurally

- Typical **raise** mechanism

```
begin
  ...
exception
  when others =>
    Cleanup;
    raise;
end;
```

- Procedural **raise** mechanism

```
begin
  ...
exception
  when X : others =>
    Cleanup;
    Ada.Exceptions.Reraise_Occurrence (X);
end;
```

Copying **Exception_Occurrence** Objects

- Via procedure **Save_Occurrence**
  - No assignment operation since is a `limited` type

```
Error : Exception_Occurrence;

begin
   ...
exception
   when X : others =>
     Cleanup;
     Ada.Exceptions.Save_Occurrence (X, Target => Error);
end;
```

Re-Raising Outside Dynamic Call Chain

```ada
procedure Demo is
  package Exceptions is new
      Limited_Ended_Lists (Exception_Occurrence,
                           Save_Occurrence);
  Errors : Exceptions.List;
  Iteration : Exceptions.Iterator;
  procedure Normal_Processing
      (Troubles : in out Exceptions.List) is ...
begin
  Normal_Processing (Errors);
  Iteration.Initialize (Errors);
  while Iteration.More loop
    declare
      Next_Error : Exception_Occurrence;
    begin
      Iteration.Read (Next_Error);
      Put_Line (Exception_Information (Next_Error));
      if Exception_Identity (Next_Error) =
         Trouble.Fatal_Error'Identity
      then
        Reraise_Occurrence (Next_Error);
      end if;
    end;
  end loop;
  Put_Line ("Done");
end Demo;
```

In Practice

Fulfill Interface Promises To Clients

- If handled and not re-raised, normal processing continues at point of client's call
- Hence caller expectations must be satisfied

```
procedure Get (Reading : out Sensor_Reading) is
begin
  ...
  Reading := New_Value;
  ...
exceptions
  when Some_Error =>
    Reading := Default_Value;
end Get;

function Foo return Some_Type is
begin
  ...
  return Determined_Value;
  ...
exception
  when Some_Error =>
    return Default_Value; -- error if this isn't here
end Foo;
```

Allow Clients To Avoid Exceptions

- Callee

```
package Stack is
  Overflow : exception;
  Underflow : exception;
  function Full return Boolean;
  function Empty return Boolean;
  procedure Push (Item : in Some_Type);
  procedure Pop (Item : out Some_Type);
end Stack;
```

- Caller

```
if not Stack.Empty then
  Stack.Pop( ... );  -- will not raise Underflow
```

You Can Suppress Run-Time Checks

- Syntax (could use a compiler switch instead)

```
pragma Suppress ( check-name [, [On =>] name] );
```

- Language-defined checks emitted by compiler
- Compiler may ignore request if unable to comply
- Behavior will be unpredictable if exceptions occur
  - Raised within the region of suppression
  - Propagated into region of suppression

```
pragma Suppress (Range_Check);
pragma Suppress (Index_Check, On => Table);
```

Error Classifications

- Some errors must be detected at run-time
  - Corresponding to the predefined exceptions
- **Bounded Errors**
  - Need not be detected prior to/during execution if too hard
  - If not detected, range of possible effects is bounded
    - Possible effects are specified per error
  - Example: evaluating an un-initialized scalar variable
  - It might "work"!
- **Erroneous Execution**
  - Need not be detected prior to/during execution if too hard
  - If not detected, range of possible effects is not bounded
  - Example: Occurrence of a suppressed check

Lab

Advanced Exceptions Lab

(Simplified) Calculator

- Overview
  - Create an application that allows users to enter a simple calculation and get a result
- Goal
  - Application should allow user to add, subtract, multiply, and divide
  - We want to track exceptions without actually "interrupting" the application
  - When the user has finished entering data, the application should report the errors found

Project Requirements

- Exception Tracking
  - Input errors should be flagged (e.g. invalid operator, invalid numbers)
  - Divide by zero should be it's own special case exception
  - Operational errors (overflow, etc) should be flagged in the list of errors
- Driver
  - User should be able to enter a string like "1 + 2" and the program will print "3"
  - User should not be interrupted by error messages
  - When user is done entering data, print all errors (raised exceptions)
- Extra Credit
  - Allow multiple operations on a line

Advanced Exceptions Lab Solution - Calculator (Spec)

```
package Calculator is
   Formatting_Error : exception;
   Divide_By_Zero   : exception;
   type Integer_T is range -1_000 .. 1_000;
   function Add
     (Left, Right : String)
      return Integer_T;
   function Subtract
     (Left, Right : String)
      return Integer_T;
   function Multiply
     (Left, Right : String)
      return Integer_T;
   function Divide
     (Top, Bottom : String)
      return Integer_T;
end Calculator;
```

Advanced Exceptions Lab Solution - Main

```ada
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;
with Calculator;            use Calculator;
with Debug_Pkg;
with Input;                 use Input;
procedure Main is
   Illegal_Operator : exception;
   procedure Parser
     (Str      :     String;
      Left     : out Unbounded_String;
      Operator : out Unbounded_String;
      Right    : out Unbounded_String) is
      I : Integer := Str'first;
   begin
      while I <= Str'length and then Str (I) /= ' ' loop
         Left := Left & Str (I);
         I    := I + 1;
      end loop;
      while I <= Str'length and then Str (I) = ' ' loop
         I := I + 1;
      end loop;
      while I <= Str'length and then Str (I) /= ' ' loop
         Operator := Operator & Str (I);
         I        := I + 1;
      end loop;
      while I <= Str'length and then Str (I) = ' ' loop
         I := I + 1;
      end loop;
      while I <= Str'length and then Str (I) /= ' ' loop
         Right := Right & Str (I);
         I     := I + 1;
      end loop;
   end Parser;
begin
   loop
      declare
         Left, Operator, Right : Unbounded_String;
         Input                 : constant String := Get_String ("Sequence");
      begin
         exit when Input'length = 0;
         Parser (Input, Left, Operator, Right);
         case Element (Operator, 1) is
            when '+' =>
               Put_Line
                 ("  => " &
                  Integer_T'image (Add (To_String (Left), To_String (Right))));
            when '-' =>
               Put_Line
                 ("  => " &
                  Integer_T'image
                    (Subtract (To_String (Left), To_String (Right))));
            when '*' =>
               Put_Line
                 ("  => " &
                  Integer_T'image
                    (Multiply (To_String (Left), To_String (Right))));
            when '/' =>
               Put_Line
                 ("  => " &
                  Integer_T'image
                    (Divide (To_String (Left), To_String (Right))));
            when others =>
               raise Illegal_Operator;
         end case;
      exception
         when The_Err : others =>
            Debug_Pkg.Save_Occurrence (The_Err);
      end;
   end loop;
   Debug_Pkg.Print_Exceptions;
end Main;
```

Advanced Exceptions Lab Solution - Calculator (Body)

```
package body Calculator is
   function Value
     (Str : String)
      return Integer_T is
   begin
      return Integer_T'value (Str);
   exception
      when Constraint_Error =>
         raise Formatting_Error;
   end Value;
   function Add
     (Left, Right : String)
      return Integer_T is
   begin
      return Value (Left) + Value (Right);
   end Add;
   function Subtract
     (Left, Right : String)
      return Integer_T is
   begin
      return Value (Left) - Value (Right);
   end Subtract;
   function Multiply
     (Left, Right : String)
      return Integer_T is
   begin
      return Value (Left) * Value (Right);
   end Multiply;
   function Divide
     (Top, Bottom : String)
      return Integer_T is
   begin
      if Value (Bottom) = 0 then
         raise Divide_By_Zero;
      else
         return Value (Top) / Value (Bottom);
      end if;
   end Divide;
end Calculator;
```

Advanced Exceptions Lab Solution - Debug

```
with Ada.Exceptions;
package Debug_Pkg is
   procedure Save_Occurrence (X : Ada.Exceptions.Exception_Occurrence);
   procedure Print_Exceptions;
end Debug_Pkg;

with Ada.Exceptions;
with Ada.Text_IO;
use type Ada.Exceptions.Exception_Id;
package body Debug_Pkg is
   Exceptions     : array (1 .. 100) of Ada.Exceptions.Exception_Occurrence;
   Next_Available : Integer := 1;
   procedure Save_Occurrence (X : Ada.Exceptions.Exception_Occurrence) is
   begin
      Ada.Exceptions.Save_Occurrence (Exceptions (Next_Available), X);
      Next_Available := Next_Available + 1;
   end Save_Occurrence;
   procedure Print_Exceptions is
   begin
      for I in 1 .. Next_Available - 1 loop
         declare
            E    : Ada.Exceptions.Exception_Occurrence renames Exceptions (I);
            Flag : Character := ' ';
         begin
            if Ada.Exceptions.Exception_Identity (E) =
              Constraint_Error'identity
            then
               Flag := '*';
            end if;
            Ada.Text_IO.Put_Line
              (Flag & " " & Ada.Exceptions.Exception_Information (E));
         end;
      end loop;
   end Print_Exceptions;
end Debug_Pkg;
```

Summary

Summary

- Re-raising exceptions is possible
- Suppressing checks is allowed but requires care
    - Testing only proves presence of errors, not absence
    - Exceptions may occur anyway, with unpredictable effects

Advanced Tasking

Introduction

A Simple Task

- Parallel code execution via **task**
- `limited` types (No copies allowed)

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

```
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**

Rendezvous Calls

- When calling an **entry**, the caller waits until the task is ready to be called

```
Put_Line ("calling start");
T.Start;
Put_Line ("calling receive 1");
T.Receive_Message ("1");
Put_Line ("calling receive 2");
-- Locks until somebody calls Start
T.Receive_Message ("2");
```

- Results in an output like:

```
calling start
start
calling receive 1
Receive 1
calling receive 2
```

Accepting a Rendezvous

- Simple `accept` statement
  - Used by a server task to indicate a willingness to provide the service at a given point
- Selective `accept` statement (later in these slides)
  - Wait for more than one rendezvous at any time
  - Time-out if no rendezvous within a period of time
  - Withdraw its offer if no rendezvous is immediately available
  - Terminate if no clients can possibly call its entries
  - Conditionally accept a rendezvous based on a guard expression

Example: Task - Declaration

```
package Tasks is

   task T is
      entry Start;
      entry Receive_Message (V : String);
   end T;

end Tasks;
```

Example: Task - Body

```
with Ada.Text_IO; use Ada.Text_IO;

package body Tasks is

   task body T is
   begin
      loop
         accept Start do
            Put_Line ("Start");
         end Start;

         accept Receive_Message (V : String) do
            Put_Line ("Receive " & V);
         end Receive_Message;
      end loop;
   end T;

end Tasks;
```

Example: Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Tasks;       use Tasks;

procedure Main is
begin
   Put_Line ("calling start");
   T.Start;
   Put_Line ("calling receive 1");
   T.Receive_Message ("1");
   Put_Line ("calling receive 2");
   --  Locks until somebody calls Start
   T.Receive_Message ("2");
end Main;
```

Quiz

```
task type T is
    entry Go;
end T;

task body T is
begin
    accept Go do
        loop
            null;
        end loop;
    end Go;
end T;

My_Task : T;
```

What happens when `My_Task.Go` is called?

A. Compilation error
B. Runtime error
C. The calling task hangs
D. My_Task hangs

Quiz

```
task type T is
    entry Go;
end T;

task body T is
begin
    accept Go do
        loop
            null;
        end loop;
    end Go;
end T;

My_Task : T;
```

What happens when `My_Task.Go` is called?

A. Compilation error
B. Runtime error
C. ***The calling task hangs***
D. ***My_Task hangs***

Quiz

```
task type T is
    entry Go;
end T;

task body T is
begin
    accept Go;
    loop
        null;
    end loop;
end T;
```

```
My_Task : T;
```

What happens when `My_Task.Go` is called?

A. Compilation error
B. Runtime error
C. The calling task hangs
D. My_Task hangs

Quiz

```
task type T is
    entry Go;
end T;

task body T is
begin
    accept Go;
    loop
        null;
    end loop;
end T;
```

```
My_Task : T;
```

What happens when `My_Task.Go` is called?

A. Compilation error
B. Runtime error
C. The calling task hangs
D. ***My_Task hangs***

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   task T is
      entry Hello;
      entry Goodbye;
   end T;
   task body T is
   begin
      loop
         accept Hello do
            Put_Line ("Hello");
         end Hello;
         accept Goodbye do
            Put_Line ("Goodbye");
         end Goodbye;
      end loop;
      Put_Line ("Finished");
   end T;
begin
   T.Hello;
   T.Goodbye;
   Put_Line ("Done");
end Main;
```

What is the output of this program?

A. Hello, Goodbye, Finished, Done
B. Hello, Goodbye, Finished
C. Hello, Goodbye, Done
D. Hello, Goodbye

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   task T is
      entry Hello;
      entry Goodbye;
   end T;
   task body T is
   begin
      loop
         accept Hello do
            Put_Line ("Hello");
         end Hello;
         accept Goodbye do
            Put_Line ("Goodbye");
         end Goodbye;
      end loop;
      Put_Line ("Finished");
   end T;
begin
   T.Hello;
   T.Goodbye;
   Put_Line ("Done");
end Main;
```

What is the output of this program?

A. Hello, Goodbye, Finished, Done
B. Hello, Goodbye, Finished
C. ***Hello, Goodbye, Done***
D. Hello, Goodbye

- Entries `Hello` and `Goodbye` are reached (so "Hello" and "Goodbye" are printed).
- After `Goodbye`, task returns to `Main` (so "Done" is printed) but the loop in the task never finishes (so "Finished" is never printed).

Protected Objects

Protected Objects

- **Multitask-safe** accessors to get and set state
- **No** direct state manipulation
- **No** concurrent modifications
- `limited` types (No copies allowed)

Protected: Functions and Procedures

- A `function` can **get** the state
  - **Multiple-Readers**
  - Protected data is **read-only**
  - Concurrent call to `function` is **allowed**
  - **No** concurrent call to `procedure`
- A `procedure` can **set** the state
  - **Single-Writer**
  - **No** concurrent call to either `procedure` or `function`
- In case of concurrency, other callers get **blocked**
  - Until call finishes
- Support for read-only locks **depends on OS**
  - Windows has **no** support for those
  - In that case, `function` are **blocking** as well

Protected: Limitations

- **No** potentially blocking action
  - `select`, `accept`, `entry` call, `delay`, `abort`
  - `task` creation or activation
  - Some standard lib operations, eg. IO
    - Depends on implementation
- May raise `Program_Error` or deadlocks
- **Will** cause performance and portability issues
- `pragma Detect_Blocking` forces a proactive runtime detection
- Solve by deferring blocking operations
  - Using eg. a FIFO

Protected: Lock-Free Implementation

- GNAT-Specific
- Generates code without any locks
- Best performance
- No deadlock possible
- Very constrained
  - No reference to entities **outside** the scope
  - No direct or indirect `entry`, `goto`, `loop`, `procedure` call
  - No `access` dereference
  - No composite parameters
  - See GNAT RM 2.100

```
protected Object
   with Lock_Free is
```

Example: Protected Objects - Declaration

```
package Protected_Objects is

   protected Object is

      procedure Set (Prompt : String; V : Integer);
      function Get (Prompt : String) return Integer;

   private
      Local : Integer := 0;
   end Object;

end Protected_Objects;
```

Example: Protected Objects - Body

```
with Ada.Text_IO; use Ada.Text_IO;

package body Protected_Objects is

   protected body Object is

      procedure Set (Prompt : String; V : Integer) is
         Str : constant String := "Set " & Prompt & V'Image;
      begin
         Local := V;
         Put_Line (Str);
      end Set;

      function Get (Prompt : String) return Integer is
         Str : constant String := "Get " & Prompt & Local'Image;
      begin
         Put_Line (Str);
         return Local;
      end Get;

   end Object;

end Protected_Objects;
```

Quiz

```
procedure Main is
    protected type O is
       entry P;
    end O;

    protected body O is
       entry P when True is
       begin
          Put_Line ("OK");
       end P;
    end O;
begin
    O.P;
end Main;
```

What is the result of compiling and running this code?

A. "OK"
B. Nothing
C. Compilation error
D. Runtime error

Quiz

```
procedure Main is
    protected type O is
       entry P;
    end O;

    protected body O is
       entry P when True is
       begin
          Put_Line ("OK");
       end P;
    end O;
begin
    O.P;
end Main;
```

What is the result of compiling and running this code?

A. "OK"
B. Nothing
C. ***Compilation error***
D. Runtime error

`O` is a `protected type`, needs instantiation

Quiz

```
protected O is
   function Get return Integer;
   procedure Set (V : Integer);
private
   Val, Access_Count : Integer := 0;
end O;

protected body O is
   function Get return Integer is
   begin
      Access_count := Access_Count + 1;
      return Val;
   end Get;

   procedure Set (V : Integer) is
   begin
      Access_count := Access_Count + 1;
      Val := V;
   end Set;
end O;
```

What is the result of compiling and running this code?

A. No error
B. Compilation error
C. Runtime error

Quiz

```
protected O is
   function Get return Integer;
   procedure Set (V : Integer);
private
   Val, Access_Count : Integer := 0;
end O;

protected body O is
   function Get return Integer is
   begin
      Access_count := Access_Count + 1;
      return Val;
   end Get;

   procedure Set (V : Integer) is
   begin
      Access_count := Access_Count + 1;
      Val := V;
   end Set;
end O;
```

What is the result of compiling and running this code?

A. No error
B. ***Compilation error***
C. Runtime error

Cannot set `Access_Count` from a `function`

Quiz

```
protected P is
   procedure Initialize (V : Integer);
   procedure Increment;
   function Decrement return Integer;
   function Query return Integer;
private
   Object : Integer := 0;
end P;
```

What of the following completions for P's members is illegal?

A.
```
procedure Initialize (V : Integer) is
begin
  Object := V;
end Initialize;
```

B.
```
procedure Increment is
begin
  Object := Object + 1;
end Increment;
```

C.
```
function Decrement return Integer is
begin
  Object := Object - 1;
  return Object;
end Decrement;
```

D.
```
function Query return Integer is begin
  return Object;
end Query;
```

Quiz

```
protected P is
   procedure Initialize (V : Integer);
   procedure Increment;
   function Decrement return Integer;
   function Query return Integer;
private
   Object : Integer := 0;
end P;
```

What of the following completions for P's members is illegal?

A.
```
procedure Initialize (V : Integer) is
begin
  Object := V;
end Initialize;
```

B.
```
procedure Increment is
begin
  Object := Object + 1;
end Increment;
```

C.
```
function Decrement return Integer is
begin
  Object := Object - 1;
  return Object;
end Decrement;
```

D.
```
function Query return Integer is begin
  return Object;
end Query;
```

A. Legal
B. Legal - subprograms do not need parameters
C. Functions in a protected object cannot modify global objects
D. Legal

Delays

Delay keyword

- **delay** keyword part of tasking
- Blocks for a time
- Relative: Blocks for at least `Duration`
- Absolute: Blocks until no earlier than `Calendar.Time` or `Real_Time.Time`

```
with Calendar;

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

Task and Protected Types

Task Activation

- Instantiated tasks start running when **activated**
- On the **stack**
  - When **enclosing** declarative part finishes **elaborating**
- On the **heap**
  - **Immediately** at instantiation

```
task type First_T is ...
type First_T_A is access all First_T;

task body First_T is ...
...
declare
   V1 : First_T;
   V2 : First_T_A;
begin  -- V1 is activated
   V2 := new First_T;  -- V2 is activated immediately
```

Single Declaration

- Instantiate an **anonymous** task (or protected) type
- Declares an object of that type

```
task type Task_T is
   entry Start;
end Task_T;

type Task_Ptr_T is access all Task_T;

task body Task_T is
begin
   accept Start;
end Task_T;
...
   V1 : Task_T;
   V2 : Task_Ptr_T;
begin
   V1.Start;
   V2 := new Task_T;
   V2.all.Start;
```

Task Scope

- Nesting is possible in **any** declarative block
- Scope has to **wait** for tasks to finish before ending
- At library level: program ends only when **all tasks** finish

```
package P is
   task T;
end P;

package body P is
   task body T is
      loop
         delay 1.0;
         Put_Line ("tick");
      end loop;
   end T;
end P;
```

Waiting On Different Entries

- It is convenient to be able to accept several entries
- The **select** statements can wait simultaneously on a list of entries
  - For **task** only
  - It accepts the **first** one that is requested

```
select
  accept Receive_Message (V : String)
  do
    Put_Line ("Message : " & String);
  end Receive_Message;
or
  accept Stop;
    exit;
  end select;
```

Example: Protected Objects - Declaration

```
package Protected_Objects is

   protected type Object is
      procedure Set (Caller : Character; V : Integer);
      function Get return Integer;
      procedure Initialize (My_Id : Character);

   private

      Local : Integer   := 0;
      Id    : Character := ' ';
   end Object;

   O1, O2 : Object;

end Protected_Objects;
```

Example: Protected Objects - Body

```
with Ada.Text_IO; use Ada.Text_IO;

package body Protected_Objects is

   protected body Object is

      procedure Initialize (My_Id : Character) is
      begin
         Id := My_Id;
      end Initialize;

      procedure Set (Caller : Character; V : Integer) is
      begin
         Local := V;
         Put_Line ( "Task-" & Caller & " Object-" & Id & " => " & V'Image );
      end Set;

      function Get return Integer is
      begin
         return Local;
      end Get;
   end Object;

end Protected_Objects;
```

Example: Tasks - Declaration

```
package Tasks is
   task type T is
      entry Start
        (Id : Character; Initial_1, Initial_2 : Integer);
      entry Receive_Message (Delta_1, Delta_2 : Integer);
   end T;

   T1, T2 : T;
end Tasks;
```

Example: Tasks - Body

```
task body T is
   My_Id : Character := ' ';
   ...
   accept Start (Id : Character; Initial_1, Initial_2 : Integer) do
      My_Id := Id;
      O1.Set (My_Id, Initial_1);
      O2.Set (My_Id, Initial_2);
   end Start;

   loop
      accept Receive_Message (Delta_1, Delta_2 : Integer) do
         declare
            New_1 : constant Integer := O1.Get + Delta_1;
            New_2 : constant Integer := O2.Get + Delta_2;
         begin
            O1.Set (My_Id, New_1);
            O2.Set (My_Id, New_2);
         end;
      end Receive_Message;
   end loop;
```

Example: Main

```
with Tasks;             use Tasks;
with Protected_Objects; use Protected_Objects;

procedure Test_Protected_Objects is
begin
   O1.Initialize ('X');
   O2.Initialize ('Y');
   T1.Start ('A', 1, 2);
   T2.Start ('B', 1_000, 2_000);
   T1.Receive_Message (1, 2);
   T2.Receive_Message (10, 20);

   --  Ugly...
   abort T1;
   abort T2;
end Test_Protected_Objects;
```

Some Advanced Concepts

Waiting With a Delay

- A **select** statement can wait with a **delay**
  - If that delay is exceeded with no entry call, block is executed
- The **delay until** statement can be used as well
- There can be multiple **delay** statements
  - (useful when the value is not hard-coded)

```
select
  accept Receive_Message (V:String) do
    Put_Line ("Message : " & String);
  end Receive_Message;
or
  delay 50.0;
  Put_Line ("Don't wait any longer");
  exit;
end select;
```

Calling an Entry With a Delay Protection

- A call to `entry` **blocks** the task until the entry is `accept` 'ed
- Wait for a **given amount of time** with `select ... delay`
- Only **one** entry call is allowed
- No `accept` statement is allowed

```
task Msg_Box is
   entry Receive_Message (V : String);
end Msg_Box;

procedure Main is
begin
   select
      Msg_Box.Receive_Message ("A");
   or
      delay 50.0;
   end select;
end Main;
```

The Delay Is Not A Timeout

- The time spent by the client is actually **not bounded**
  - Delay's timer **stops** on `accept`
  - The call blocks **until** `end` of server-side statements
- In this example, the total delay is up to **1010 s**

```
task body Msg_Box is
   accept Receive_Message (S : String) do
      delay 1000.0;
   end Receive_Message;
...
procedure Client is
begin
   select
      Msg_Box.Receive_Message ("My_Message")
   or
      delay 10.0;
   end select;
```

Non-blocking Accept or Entry

- Using **else**
    - Task **skips** the **accept** or **entry** call if they are **not ready** to be entered
- On an **accept**

```
select
   accept Receive_Message (V : String) do
      Put_Line ("T: Receive " & V);
   end Receive_Message;
else
   Put_Line ("T: Nothing received");
end select;
```

- As caller on an **entry**

```
select
   T.Stop;
else
   Put_Line ("No stop");
end select;
```

- **delay** is **not** allowed in this case

Issues With "Double Non-Blocking"

- For `accept ... else` the server **peeks** into the queue
  - Server **does not** wait
- For `<entry-call> ... else` the caller looks for a **waiting** server
- If both use it, the entry will **never** be called
- Server

```
select
   accept Receive_Message (V : String) do
      Put_Line ("T: Receive " & V);
   end Receive_Message;
else
   Put_Line ("T: Nothing received");
end select;
```

- Caller

```
select
   T.Receive_Message ("1");
else
   Put_Line ("No message sent");
end select;
```

Terminate Alternative

- An entry can't be called anymore if all tasks calling it are over
- Handled through `or terminate` alternative
  - Terminates the task if **all others** are terminated
  - Or are **blocked** on `or terminate` themselves
- Task is terminated **immediately**
  - No additional code executed

```
select
   accept Entry_Point
or
   terminate;
end select;
```

Guard Expressions

- **accept** may depend on a **guard condition** with **when**
  - Evaluated when entering **select**

```
task body T is
   Val : Integer;
   Initialized : Boolean := False;
begin
   loop
      select
         accept Put (V : Integer) do
            Val := V;
            Initialized := True;
         end Put;
      or
         when Initialized =>
            accept Get (V : out Integer) do
               V := Val;
            end Get;
      end select;
   end loop;
end T;
```

Protected Object Entries

- **Special** kind of protected `procedure`
- May use a **barrier**, that **only** allows call on a **boolean** condition
- Barrier is **evaluated** and may be **relieved** when
  - A task calls `entry`
  - A protected `entry` or `procedure` is **exited**
- Several tasks can be waiting on the same `entry`
  - Only **one** will be re-activated when the barrier is relieved

```
protected body Stack is
   entry Push (V : Integer) when Size < Buffer'Length is
   ...
   entry Pop  (V : out Integer) when Size > 0 is
   ...
end Object;
```

Select On Protected Objects Entries

- Same as **select** but on task entries
  - With a **delay** part

```
select
   O.Push (5);
or
   delay 10.0;
   Put_Line ("Delayed overflow");
end select;
```

  - or with an **else** part

```
select
   O.Push (5);
else
   Put_Line ("Overflow");
end select;
```

Queue

- Protected `entry`, `procedure`, and tasks `entry` are activated by **one** task at a time
- **Mutual exclusion** section
- Other tasks trying to enter are **queued**
    - In **First-In First-Out** (FIFO) by default
- When the server task **terminates**, tasks still queued receive `Tasking_Error`

Queuing Policy

- Queuing policy can be set using

```
pragma Queuing_Policy (<policy_identifier>);
```

- The following policy_identifier are available
  - FIFO_Queuing (default)
  - Priority_Queuing
- FIFO_Queuing
  - First-in First-out, classical queue
- Priority_Queuing
  - Takes into account priority
  - Priority of the calling task **at time of call**

Setting Task Priority

- GNAT available priorities are 0 .. 30, see **gnat/system.ads**
- Tasks with the highest priority are prioritized more
- Priority can be set **statically**

```
task T
   with Priority => <priority_level>
   is ...
```

- Priority can be set **dynamically**

```
with Ada.Dynamic_Priorities;

task body T is
begin
   Ada.Dynamic_Priorities.Set_Priority (10);
end T;
```

requeue Instruction

- requeue can be called in any entry (task or protected)
- Puts the requesting task back into the queue
  - May be handled by another entry
  - Or the same one...
- Reschedule the processing for later

```
entry Extract (Qty : Integer) when True is
begin
   if not Try_Extract (Qty) then
      requeue Extract;
   end if;
end Extract;
```

- Same parameter values will be used on the queue

`requeue` Tricks

- Only an accepted call can be requeued
- Accepted entries are waiting for `end`
  - Not in a `select ... or delay ... else` anymore
- So the following means the client blocks for **2 seconds**

```
task body Select_Requeue_Quit is
begin
   accept Receive_Message (V : String) do
      requeue Receive_Message;
   end Receive_Message;
   delay 2.0;
end Select_Requeue_Quit;
   ...
   select
      Select_Requeue_Quit.Receive_Message ("Hello");
   or
      delay 0.1;
   end select;
```

Abort Statements

- `abort` stops the tasks **immediately**
  - From an external caller
  - No cleanup possible
  - Highly unsafe - should be used only as **last resort**

```
procedure Main is
   task T;

   task T is
   begin
      loop
         delay 1.0;
         Put_Line ("A");
      end loop;
   end T;
begin
   delay 10.0;
   abort T;
end;
```

`select` ... `then abort`

- `select` can call `abort`
- Can abort anywhere in the processing
- **Highly** unsafe

Multiple Select Example

```
loop
   select
      accept Receive_Message (V : String) do
         Put_Line ("Select_Loop_Task Receive: " & V);
      end Receive_Message;
   or
      accept Send_Message (V : String) do
         Put_Line ("Select_Loop_Task Send: " & V);
      end Send_Message;
   or when Termination_Flag =>
      accept Stop;
   or
      delay 0.5;
      Put_Line
        ("No more waiting at" & Day_Duration'Image (Seconds (Clock)));
      exit;
   end select;
end loop;
```

Example: Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Task_Select; use Task_Select;

procedure Main is
begin
   Select_Loop_Task.Receive_Message ("1");
   Select_Loop_Task.Send_Message ("A");
   Select_Loop_Task.Send_Message ("B");
   Select_Loop_Task.Receive_Message ("2");
   Select_Loop_Task.Stop;
exception
   when Tasking_Error =>
      Put_Line ("Expected exception: Entry not reached");
end Main;
```

Quiz

```
task T is
   entry E1;
   entry E2;
end T;
...
task body Other_Task is
begin
   select
      T.E1;
   or
      T.E2;
   end select;
end Other_Task;
```

What is the result of compiling and running this code?

A. `T.E1` is called
B. Nothing
C. Compilation error
D. Runtime error

Quiz

```
task T is
   entry E1;
   entry E2;
end T;
...
task body Other_Task is
begin
   select
      T.E1;
   or
      T.E2;
   end select;
end Other_Task;
```

What is the result of compiling and running this code?

A. `T.E1` is called
B. Nothing
C. ***Compilation error***
D. Runtime error

A `select` entry call can only call one `entry` at a time.

Quiz

```
procedure Main is
   task T is
      entry A;
   end T;

   task body T is
   begin
      select
         accept A;
         Put ("A");
      else
         delay 1.0;
      end select;
   end T;
begin
   select
      T.A;
   else
      delay 1.0;
   end select;
end Main;
```

What is the output of this code?

A. "AAAAA..."
B. Nothing
C. Compilation error
D. Runtime error

Quiz

```
procedure Main is
   task T is
      entry A;
   end T;

   task body T is
   begin
      select
         accept A;
         Put ("A");
      else
         delay 1.0;
      end select;
   end T;
begin
   select
      T.A;
   else
      delay 1.0;
   end select;
end Main;
```

What is the output of this code?

A. "AAAAA..."
B. ***Nothing***
C. Compilation error
D. Runtime error

Common mistake: `Main` and `T` won't wait on each other and will both execute their `delay` statement only.

Quiz

```
procedure Main is
   task type T is
      entry A;
   end T;

   task body T is
   begin
      select
         accept A;
      or
         terminate;
      end select;

      Put_Line ("Terminated");
   end T;

   My_Task : T;
begin
   null;
end Main;
```

What is the output of this code?

A. "Terminated"
B. Nothing
C. Compilation error
D. Runtime error

Quiz

```
procedure Main is
   task type T is
      entry A;
   end T;

   task body T is
   begin
      select
         accept A;
      or
         terminate;
      end select;

      Put_Line ("Terminated");
   end T;

   My_Task : T;
begin
   null;
end Main;
```

What is the output of this code?

A. "Terminated"
B. ***Nothing***
C. Compilation error
D. Runtime error

`T` is terminated at the end of `Main`

Quiz

```
procedure Main is
begin
   select
      delay 2.0;
   then abort
      loop
         delay 1.5;
         Put ("A");
      end loop;
   end select;

   Put ("B");
end Main;
```

What is the output of this code?

A. "A"
B. "AAAA..."
C. "AB"
D. Compilation error
E. Runtime error

Quiz

```
procedure Main is
begin
   select
      delay 2.0;
   then abort
      loop
         delay 1.5;
         Put ("A");
      end loop;
   end select;

   Put ("B");
end Main;
```

What is the output of this code?

A. "A"
B. "AAAA..."
C. ***"AB"***
D. Compilation error
E. Runtime error

`then abort` aborts the select only, not `Main`.

Quiz

```
procedure Main is
    Ok : Boolean := False

    protected O is
       entry P;
    end O;

    protected body O is
    begin
       entry P when Ok is
          Put_Line ("OK");
       end P;
    end O;
begin
    O.P;
end Main;
```

What is the result of compiling and running this code?

A. "OK"
B. Nothing
C. Compilation error
D. Runtime error

Quiz

```
procedure Main is
    Ok : Boolean := False

    protected O is
       entry P;
    end O;

    protected body O is
    begin
       entry P when Ok is
          Put_Line ("OK");
       end P;
    end O;
begin
    O.P;
end Main;
```

What is the result of compiling and running this code?

A. "OK"
B. ***Nothing***
C. Compilation error
D. Runtime error

Stuck on waiting for `Ok` to be set, `Main` will never terminate.

Standard "Embedded" Tasking Profiles

- Better performances but more constrained
- Ravenscar profile
  - Ada 2005
  - No `select`
  - No `entry` for tasks
  - Single `entry` for `protected` types
  - No entry queues
- Jorvik profile
  - Ada 2022
  - Less constrained, still performant
  - Any number of `entry` for `protected` types
  - Entry queues
- See RM D.13

Summary

Summary

- Tasks are language-based multithreading mechanisms
    - Not necessarily designed to be operated in parallel
    - Original design assumed task-switching / time-slicing
- Multiple mechanisms to synchronize tasks
    - Delay
    - Rendezvous
    - Protected Objects

Low Level Programming

Introduction

Introduction

- Sometimes you need to get your hands dirty
- Hardware Issues
  - Register or memory access
  - Assembler code for speed or size issues
- Interfacing with other software
  - Object sizes
  - Endianness
  - Data conversion

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 My_Int is range 1 .. 10;
for My_Int'Object_Size use 8;
for My_Int'Value_Size  use 4;
for My_Int'Alignment   use 1;

-- using Ada 2012 aspects
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;
for T1'Size use 3;

-- using Ada 2012 aspects
type T2 is range 1 .. 4
   with Size => 3;
```

Object Size (GNAT-Specific)

- **Object_Size** represents the size of the object in memory
- It must be a multiple of **Alignment * Storage_Unit (8)**, and at least equal to **Size**

```
type T1 is range 1 .. 4;
for T1'Value_Size use 3;
for T1'Object_Size use 8;

-- using Ada 2012 aspects
type T2 is range 1 .. 4
   with Value_Size  => 3,
        Object_Size => 8;
```

- Object size is the *default* size of an object, can be changed if specific representations are given

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

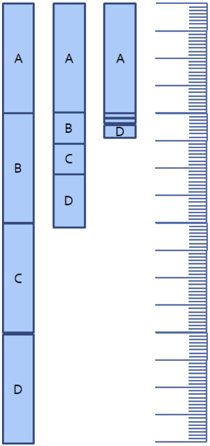
```
type T1 is range 1 .. 4;
for T1'Size use 4;
for T1'Alignment use 8;

-- using Ada 2012 aspects
type T2 is range 1 .. 4
   with Size      => 4,
        Alignment => 8;
```

Record Types

- Ada doesn't force any particular memory layout
- Depending on optimization of constraints, layout can be optimized for speed, size, or not optimized

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record;
```



Pack Aspect

- **pack** aspect (or pragma) 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 : Boolean;
   D : Enum;
end record;
type Ar is array (1 .. 1000) of Boolean;
-- Rec'Size is 56, Ar'Size is 8000
```

- Packed

```
type Enum is (E1, E2, E3);
type Rec is record
   A : Integer;
   B : Boolean;
   C : Boolean;
   D : Enum;
end record with Pack;
type Ar is array (1 .. 1000) of Boolean;
pragma Pack (Ar);
-- Rec'Size is 36, Ar'Size is 1000
```

Record Representation Clauses

- The developer can specify the exact mapping between a record and its binary representation
- This mapping can be used for optimization purposes, or to match hardware requirements
  - driver mapped on the address space, communication protocol, binary file representation...
- 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

- The size of an array component can be specified with the **Component_Size** aspect (or attribute)

```
type Ar1 is array (1 .. 1000) of Boolean;
for Ar1'Component_Size use 2;

-- using Ada 2012 aspects
type Ar2 is array (1 .. 1000) of Boolean
    with Component_Size => 2;
```

Endianness Specification (GNAT Specific)

- GNAT allows defining the endianness through the **Scalar_Storage_Order** aspect, on composite types
- Need to be associated with a consistent **Bit_Order** (convention for the bit range numbering)
- The compiler will perform bitwise transformations if needed when sending data to the processor

```
type Rec is record
   A : Integer;
   B : Boolean;
end record;
for Rec'Bit_Order use System.High_Order_First;
for Rec'Scalar_Storage_Order use System.High_Order_First;

type Ar is array (1 .. 1000) of Boolean;
for Ar'Scalar_Storage_Order use System.Low_Order_First;

-- using Ada 2012 aspects
type Rec is record
     A : Integer;
     B : Boolean;
  end record with
    Bit_Order            => High_Order_First,
    Scalar_Storage_Order => High_Order_First;

type Ar is array (1 .. 1000) of Boolean with
  Scalar_Storage_Order => Low_Order_First;
```

Change of Representation

- Explicit conversion can be used to change representation
- Very useful to unpack data from file/hardware to speed up references

```
type Rec_T is record
     Field1 : Unsigned_8;
     Field2 : Unsigned_16;
     Field3 : Unsigned_8;
end record;
type Packed_Rec_T is new Rec_T;
for Packed_Rec_T use record
   Field1 at 0 range  0 ..  7;
   Field2 at 0 range  8 .. 23;
   Field3 at 0 range 24 .. 31;
end record;
R : Rec_T;
P : Packed_Rec_T;
...
R := Rec_T (P);
P := Packed_Rec_T (R);
```

Address Clauses and Overlays

Address

- Ada distinguishes the notions of
  - A reference to an object
  - An abstract notion of address (**System.Address**)
  - The integer representation of an address
- Safety is preserved by letting the developer manipulate the right level of abstraction
- Conversion between pointers, integers and addresses are possible
- The address of an object can be specified through the **Address** aspect

Address Clauses

- Ada allows specifying the address of an entity

```
Var : Unsigned_32;
for Var'Address use ... ;
```

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

```
pragma Volatile (Var);
```

- Useful to read a value anywhere

```
function Get_Byte (Addr : Address) return Unsigned_8 is
  V : Unsigned_8;
  for V'Address use Addr;
  pragma Import (Ada, V);
begin
  return V;
end;
```

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

Address Values

- The type **Address** is declared in **System**
  - But this is a `private` type
  - You cannot use a number
- Ada standard way to set constant addresses:
  - Use **System.Storage_Elements** which allows arithmetic on address

```
for V'Address use
   System.Storage_Elements.To_Address (16#120#);
```

- GNAT specific attribute **'To_Address**
  - Handy but not portable

```
for V'Address use System'To_Address (16#120#);
```

Volatile

- The **Volatile** property can be set using an aspect (in Ada2012 only) or a pragma
- Ada also allows volatile types as well as objects.

```
type Volatile_U16 is mod 2**16;
pragma Volatile(Volatile_U16);
type Volatile_U32 is mod 2**32 with Volatile; -- Ada 201
```

- Volatile means that the exact sequence of reads and writes of an object indicated in the source code must be respected in the generated code.
  - No optimization of reads and writes please!
- Volatile types are passed by-reference.

Ada Address Example

```
type Bitfield is array (Integer range <>) of Boolean;

V  : aliased Integer; -- object can be referenced elsewhere
Pragma Volatile (V);  -- may be updated at any time

V2 : aliased Integer;
Pragma Volatile (V2);

V_A : System.Address := V'Address;
V_I : Integer_Address := To_Integer (V_A);

--  This maps directly on to the bits of V
V3 : aliased Bitfield (1 .. V'Size);
For V3'address use V_A; -- overlay

V4 : aliased Integer;
--  Trust me, I know what I'm doing, this is V2
For V4'address use To_Address (V_I - 4);
```

Aliasing Detection

- Aliasing happens when one object has two names
  - Two pointers pointing to the same object
  - Two references referencing the same object
  - Two variables at the same address
- `Var1'Has_Same_Storage (Var2)` checks if two objects occupy exactly the same space
- `Var'Overlaps_Storage (Var2)` checks if two object are partially or fully overlapping

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

Inline Assembly

Calling Assembly Code

- Calling assembly code is a vendor-specific extension
- GNAT allows passing assembly scripts directly to the linker through **System.Machine_Code.ASM**
- The developer is responsible for mapping variables on temporaries or registers
- See documentation
  - GNAT RM 13.1 Machine Code Insertion
  - GCC UG 6.39 Assembler Instructions with C Expression Operands

Simple Statement

- Instruction without inputs/outputs

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

  - Specify **Volatile** to avoid compiler optimization
  - GNAT is picky on that point

- 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

Operands

- It is often useful to have inputs or outputs...
  - **Asm_Input** and **Asm_Output** attributes on types

First operand (starting with o)

Use %% for %

Input operand for type Unsigned_8

Constraint, 'a' for eax

Value

```
Asm ("outb %0,%%dx",
     Inputs => (Unsigned_8'Asm_Input ("a", B),
                Port_Id'Asm_Input ("d", P)),
     Volatile => True);
```

Instruction

Constraint, 'd' for edx

Name

Mapping Inputs / Outputs on Temporaries

```
Asm (<script referencing $<input> >,
     Inputs  => ({<type>'Asm_Input ( <constraint>,
                                     <variable>)}),
     Outputs => ({<type>'Asm_Output ( <constraint>,
                                      <variable>)});
```

- **assembly script** containing assembly instructions + references to registers and temporaries
- **constraint** specifies how variable can be mapped on memory (see documentation for full details)

Constraint	Meaning
R	General purpose register
M	Memory
F	Floating-point register
I	A constant
D	edx (on x86)
a	eax (on x86)

Main Rules

- No control flow between assembler statements
  - Use Ada control flow statement
  - Or use control flow within one statement
- Avoid using fixed registers
  - Makes compiler's life more difficult
  - Let the compiler choose registers
  - You should correctly describe register constraints
- On x86, the assembler uses `AT&T` convention
  - First operand is source, second is destination
  - See GNU assembler manual for details

Volatile and Clobber ASM Parameters

- Volatile $\rightarrow$ True deactivates optimizations with regards to suppressed instructions
- Clobber $\rightarrow$ "reg1, reg2, ..." contains the list of registers considered to be "destroyed" by the use of the ASM call
  - Use 'memory' if the memory is accessed in an unpredictable fashion. The compiler will not keep memory values cached in registers across the instruction.

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 ("=d", Low),
            Unsigned_32'Asm_Output ("=a", High)),
        Volatile => True);
   Values := Unsigned_64 (Low) +
             Unsigned_64 (High) * 2 ** 32;
   Put_Line (Values'Img);
end Main;
```

Reading a Machine Register (ppc)

```
function Get_MSR return MSR_Type is
   Res : MSR_Type;
begin
   Asm ("mfmsr %0",
        Outputs => MSR_Type'Asm_Output ("=r", Res),
        Volatile => True);
   return Res;
end Get_MSR;
generic
    Spr : Natural;
 function Get_Spr return Unsigned_32;
 function Get_Spr return Unsigned_32 is
    Res : Unsigned_32;
 begin
    Asm ("mfspr %0,%1",
         Inputs => Natural'Asm_Input ("K", Spr),
         Outputs => Unsigned_32'Asm_Output ("=r", Res),
         Volatile => True);
    return Res;
 end Get_Spr;
 function Get_Pir is new Get_Spr (286);
```

Writing a Machine Register (ppc)

```
generic
   Spr : Natural;
procedure Set_Spr (V : Unsigned_32);
procedure Set_Spr (V : Unsigned_32) is
begin
   Asm ("mtspr %0,%1",
        Inputs => (Natural'Asm_Input ("K", Spr),
                   Unsigned_32'Asm_Input ("r", V)));
end Set_Spr;
```

Tricks

Package Interfaces

- Package **Interfaces** provide integer and unsigned types for many sizes
    - **Integer_8**, **Integer_16**, **Integer_32**, **Integer_64**
    - **Unsigned_8**, **Unsigned_16**, **Unsigned_32**, **Unsigned_64**
- With shift/rotation functions for unsigned types

Fat/Thin pointers for Arrays

- Unconstrained array access is a fat pointer

```
type String_Acc is access String;
Msg : String_Acc;
-- array bounds stored outside array pointer
```

- Use a size representation clause for a thin pointer

```
type String_Acc is access String;
for String_Acc'size use 32;
-- array bounds stored as part of array pointer
```

Flat Arrays

- A constrained array access is a thin pointer
  - No need to store bounds

```
type Line_Acc is access String (1 .. 80);
```

- You can use big flat array to index memory
  - See **GNAT.Table**
  - Not portable

```
type Char_array is array (natural) of Character;
type C_String_Acc is access Char_Array;
```

Lab

Low Level Programming Lab

(Simplified) Message generation / propagation

- Overview
  - Populate a message structure with data and a CRC (cyclic redundancy check)
  - "Send" and "Receive" messages and verify data is valid
- Goal
  - You should be able to create, "send", "receive", and print messages
  - Creation should include generation of a CRC to ensure data security
  - Receiving should include validation of CRC

Project Requirements

- Message Generation
  - Message should at least contain:
    - Unique Identifier
    - (Constrained) string field
    - Two other fields
    - CRC value
- "Send" / "Receive"
  - To simulate send/receive:
    - "Send" should do a byte-by-byte write to a text file
    - "Receive" should do a byte-by-byte read from that same text file
  - Receiver should validate received CRC is valid
    - You can edit the text file to corrupt data

Hints

- Use a representation clause to specify size of record
  - To get a valid size, individual components may need new types with their own rep spec
- CRC generation and file read/write should be similar processes
  - Need to convert a message into an array of "something"

Low Level Programming Lab Solution - CRC

```
with System;
package Crc is
   type Crc_T is mod 2**32;
   for Crc_T'size use 32;
   function Generate
     (Address : System.Address;
      Size : Natural)
      return Crc_T;
end Crc;

package body Crc is
   type Array_T is array (Positive range <>) of Crc_T;
   function Generate
     (Address : System.Address;
      Size    : Natural)
      return Crc_T is
      Word_Count : Natural;
      Retval     : Crc_T := 0;
   begin
      if Size > 0
      then
         Word_Count := Size / 32;
         if Word_Count * 32 /= Size
         then
            Word_Count := Word_Count + 1;
         end if;
         declare
            Overlay : Array_T (1 .. Word_Count);
            for Overlay'address use Address;
         begin
            for I in Overlay'range
            loop
               Retval := Retval + Overlay (I);
            end loop;
         end;
      end if;
      return Retval;
   end Generate;
end Crc;
```

Low Level Programming Lab Solution - Messages (Spec)

```
with Crc; use Crc;
package Messages is
   type Message_T is private;
   type Command_T is (Noop, Direction, Ascend, Descend, Speed);
   for Command_T use
     (Noop => 0, Direction => 1, Ascend => 2, Descend => 4, Speed => 8);
   for Command_T'size use 8;
   function Create (Command : Command_T;
                    Value   : Positive;
                    Text    : String := "")
                    return Message_T;
   function Get_Crc (Message : Message_T) return Crc_T;
   procedure Write (Message : Message_T);
   procedure Read ( Message : out Message_T;
                    valid : out boolean );
   procedure Print (Message : Message_T);
private
   type U32_T is mod 2**32;
   for U32_T'size use 32;
   Max_Text_Length : constant := 20;
   type Text_Index_T is new Integer range 0 .. Max_Text_Length;
   for Text_Index_T'size use 8;
   type Text_T is record
      Text : String (1 .. Max_Text_Length);
      Last : Text_Index_T;
   end record;
   for Text_T'size use Max_Text_Length * 8 + Text_Index_T'size;
   type Message_T is record
      Unique_Id : U32_T;
      Command   : Command_T;
      Value     : U32_T;
      Text      : Text_T;
      Crc       : Crc_T;
   end record;
end Messages;
```

Low Level Programming Lab Solution - Main (Helpers)

```
with Ada.Text_IO; use Ada.Text_IO;
with Messages;
procedure Main is
   Message : Messages.Message_T;
   function Command return Messages.Command_T is
   begin
      loop
         Put ("Command ( ");
         for E in Messages.Command_T
         loop
            Put (Messages.Command_T'image (E) & " ");
         end loop;
         Put ("): ");
         begin
            return Messages.Command_T'value (Get_Line);
         exception
            when others =>
               Put_Line ("Illegal");
         end;
      end loop;
   end Command;
   function Value return Positive is
   begin
      loop
         Put ("Value: ");
         begin
            return Positive'value (Get_Line);
         exception
            when others =>
               Put_Line ("Illegal");
         end;
      end loop;
   end Value;
   function Text return String is
   begin
      Put ("Text: ");
      return Get_Line;
   end Text;
```

Low Level Programming Lab Solution - Main

```
   procedure Create is
      C : constant Messages.Command_T := Command;
      V : constant Positive           := Value;
      T : constant String             := Text;
   begin
      Message := Messages.Create
          (Command => C,
           Value   => V,
           Text    => T);
   end Create;
   procedure Read is
      Valid : Boolean;
   begin
      Messages.Read ( Message, Valid );
      Ada.Text_IO.Put_Line("Message valid: " & Boolean'Image ( Valid ));
   end read;
begin
   loop
      Put ("Create Write Read Print: ");
      declare
         Command : constant String := Get_Line;
      begin
         exit when Command'length = 0;
         case Command (Command'first) is
            when 'c' | 'C' =>
               Create;
            when 'w' | 'W' =>
               Messages.Write (Message);
            when 'r' | 'R' =>
               read;
            when 'p' | 'P' =>
               Messages.Print (Message);
            when others =>
               null;
         end case;
      end;
   end loop;
end Main;
```

Low Level Programming Lab Solution - Messages (Helpers)

```
with Ada.Text_IO;
with Unchecked_Conversion;
package body Messages is
   Global_Unique_Id : U32_T := 0;
   function To_Text (Str : String) return Text_T is
      Length : Integer := Str'length;
      Retval : Text_T  := (Text => (others => ' '), Last => 0);
   begin
      if Str'length > Retval.Text'length then
         Length := Retval.Text'length;
      end if;
      Retval.Text (1 .. Length) := Str (Str'first .. Str'first + Length - 1);
      Retval.Last               := Text_Index_T (Length);
      return Retval;
   end To_Text;
   function From_Text (Text : Text_T) return String is
      Last : constant Integer := Integer (Text.Last);
   begin
      return Text.Text (1 .. Last);
   end From_Text;
   function Get_Crc (Message : Message_T) return Crc_T is
   begin
      return Message.Crc;
   end Get_Crc;
   function Validate (Original : Message_T) return Boolean is
      Clean : Message_T := Original;
   begin
      Clean.Crc := 0;
      return Crc.Generate (Clean'address, Clean'size) = Original.Crc;
   end Validate;
```

Low Level Programming Lab Solution - Messages (Body)

```
   function Create (Command : Command_T;
                    Value   : Positive;
                    Text    : String := "")
                    return Message_T is
      Retval : Message_T;
   begin
      Global_Unique_Id := Global_Unique_Id + 1;
      Retval           :=
        (Unique_Id => Global_Unique_Id, Command => Command,
         Value     => U32_T (Value), Text => To_Text (Text), Crc => 0);
      Retval.Crc := Crc.Generate (Retval'address, Retval'size);
      return Retval;
   end Create;
   type Char is new Character;
   for Char'size use 8;
   type Overlay_T is array (1 .. Message_T'size / 8) of Char;
   function Convert is new Unchecked_Conversion (Message_T, Overlay_T);
   function Convert is new Unchecked_Conversion (Overlay_T, Message_T);
   Const_Filename : constant String := "message.txt";
   procedure Write (Message : Message_T) is
      Overlay : constant Overlay_T := Convert (Message);
      File    : Ada.Text_IO.File_Type;
   begin
      Ada.Text_IO.Create (File, Ada.Text_IO.Out_File, Const_Filename);
      for I in Overlay'range loop
         Ada.Text_IO.Put (File, Character (Overlay (I)));
      end loop;
      Ada.Text_IO.New_Line (File);
      Ada.Text_IO.Close (File);
   end Write;
   procedure Read (Message : out Message_T;
                   Valid   : out Boolean) is
                   Overlay : Overlay_T;
                   File    : Ada.Text_IO.File_Type;
   begin
      Valid := False;
      Ada.Text_IO.Open (File, Ada.Text_IO.In_File, Const_Filename);
      declare
         Str : constant String := Ada.Text_IO.Get_Line (File);
      begin
         Ada.Text_IO.Close (File);
         for I in Str'range loop
            Overlay (I) := Char (Str (I));
         end loop;
         Message := Convert (Overlay);
         Valid   := Validate (Message);
      end;
   end Read;
   procedure Print (Message : Message_T) is
   begin
      Ada.Text_IO.Put_Line ("Message" & U32_T'image (Message.Unique_Id));
      Ada.Text_IO.Put_Line ("  " & Command_T'image (Message.Command) & " =>" &
                            U32_T'image (Message.Value));
      Ada.Text_IO.Put_Line ("  Additional Info: " & From_Text (Message.Text));
   end Print;
end Messages;
```

Summary

Summary

- Like C, Ada allows access to assembly-level programming
- Unlike C, Ada imposes some more restrictions to maintain some level of safety
- Ada also supplies language constructs and libraries to make low level programming easier